# Catalyst

## Generalised Incentives

by Ackee Blockchain

*06.05.2024*

# Contents

# 1. Document Revisions

| | | |
|---|---|---|
| 1.0 | Final report | 29.04.2024 |
| 1.1 | Fix review | 06.05.2024 |

# 2. Overview

This document presents our findings in reviewed contracts.

## 2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

## 2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

## 2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

*Low* to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

### Severity

| | | *Likelihood* | | | |
|---|---|---|---|---|---|
| | | **High** | **Medium** | **Low** | **-** |
| *Impact* | **High** | Critical | High | Medium | - |
| | **Medium** | High | Medium | Low | - |
| | **Low** | Medium | Low | Low | - |
| | **Warning** | - | - | - | Warning |
| | **Info** | - | - | - | Info |

*Table 1. Severity of findings*

## Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

## Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

## 2.4. Review team

| Member's Name | Position |
|---|---|
| Andrey Babushkin | Lead Auditor |
| Josef Gattermayer, Ph.D. | Audit Supervisor |

## 2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

# 3. Executive Summary

## Revision 1.0

Catalyst engaged Ackee Blockchain to perform a security review of the Generalised Incentives protocol with a total time donation of 10 engineering days in a period between April 15 and April 26, 2024, with Andrey Babushkin as the lead auditor.

The audit was performed on the commit e410087 [1] and the scope was the following:

- src/IncentivizedMessageEscrow.sol
- src/apps/wormhole/IncentivizedWormholeEscrow.sol
- src/apps/polymer/IncentivizedPolymerEscrow.sol

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. During the review, we paid special attention to:

- ensuring message payloads are correctly transmitted and validated,
- ensuring the arithmetic of the system is correct,
- enumerating all entry points to the contract and their possible abuse scenarios,
- validating the integration with the Wormhole and IBC protocols,
- looking for common issues such as data validation.

Our review resulted in 19 findings, ranging from Info to Critical severity. The most severe one is insufficient data validation of the message identifier (see C1). To test the arithmetics of fee calculation when a time delta is set, we performed a fuzz test using Wake, see Appendix C. This fuzz test helped to identify the floating timestamps issue, see L2.

Ackee Blockchain recommends Catalyst:

- pay special attention to data validation in the payload and input parameters,
- address the issue of non-deliverable messages and locked tokens,
- consider using the latest version of the Solidity compiler,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

## Revision 1.1

After discussing the issue with the `MessageDelivered` event it reclassified from a warning to a medium severity issue since the insufficient information in the logs could cause a Denial of Service for a specific message.

The review was done on multiple commits from several pull requests:

- PR#41, commit `d50ca3a` [2] fixes C1.
- PR#43, commit `2fbcf02` [3] fixes M1.
- PR#48, commit `cc44ec2` [4] fixes M2.
- PR#42, commit `c490e14` [5] fixes M3.
- PR#46, commit `16827be` [6] fixes L2.
- PR#45, commit `3c3bf30` [7] fixes W7.
- PR#47, commit `b4e27b2` [8] fixes W9.
- PR#44, commit `9846038` [9] fixes I2.

The main focus was on the changes made to the contracts and the fixes of the issues found in the previous review. The review was performed by Andrey Babushkin. See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope. Out of the 19 findings, 8 were fixed, and others were acknowledged.

[1] full commit hash: e410087b6faca4ce737b50a74f276e27ce7874ad

[2] full commit hash: `d50ca3a08b234f4fedb23063b41ffd05b65b52c3`

[3] full commit hash: `2fbcf02fc6ff363864301923d38274b8bc393c24`

[4] full commit hash: `cc44ec2795e783ba00e9e9eb3a22e5aea8e182ce`

[5] full commit hash: `c490e149b1c0116048f41cecf27af4503f98abb2`

[6] full commit hash: `16827beb92e8652df642c263ffe2f2c78303f4a7`

[7] full commit hash: `3c3bf300e3ff5e0ac30d197b715c7b1b13f722aa`

[8] full commit hash: `b4e27b2ef6e6c2754b64016517d470a0fb1b8276`

[9] full commit hash: `9846038457621fce58fb1303e64d5eb614579658`

# 4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

| | Severity | Reported | Status |
|---|---|---|---|
| C1: Fake escrow can craft ACK packets with any `messageIdentifier` and withdraw all bounties | Critical | 1.0 | Fixed |
| M1: Fee recipient addresses are not validated against the zero address | Medium | 1.0 | Fixed |
| M2: Insufficient validation of a disabled route may lead to the locked Ether | Medium | 1.0 | Fixed |
| M3: `MessageDelivered` event is used for both successful and failed calls | Medium | 1.0 | Fixed |

| | Severity | Reported | Status |
|---|---|---|---|
| L1: Large messages may not be delivered due to different block gas limits on different chains | Low | 1.0 | Acknowledged |
| L2: Unfair fee distribution due to floating `block.timestamp` | Low | 1.0 | Fixed |
| L3: Usage of `send` and `transfer` can make the escrow unusable for smart-contract relayers | Low | 1.0 | Acknowledged |
| W1: Usage of `solc` optimizer | Warning | 1.0 | Acknowledged |
| W2: `block.timestamp` can be different on different chains | Warning | 1.0 | Acknowledged |
| W3: Too small or too large time deltas make the fee distribution unfair | Warning | 1.0 | Acknowledged |
| W4: Setting insufficient gas for a call will lead to undelivered messages and locked assets | Warning | 1.0 | Acknowledged |
| W5: From applications are not validated for being a smart contract | Warning | 1.0 | Acknowledged |

|  | Severity | Reported | Status |
|---|---|---|---|
| W6: Paying the maximum gas fee for timeouts may incentivize relayers not to deliver messages | Warning | 1.0 | Acknowledged |
| W7: True and logged to the event gas spent values are different | Warning | 1.0 | Fixed |
| W8: Relayers are not protected against a malicious escrow on the destination chain | Warning | 1.0 | Acknowledged |
| W9: A compiler bug may create dirty storage bytes | Warning | 1.0 | Fixed |
| I1: Unused declarations | Info | 1.0 | Acknowledged |
| I2: Improve protocol documentation | Info | 1.0 | Fixed |
| I3: Use maximum line length | Info | 1.0 | Acknowledged |

*Table 2. Table of Findings*

# 5. Report revision 1.0

## 5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

The Incentivized Message Escrow protocol serves as an abstraction layer between Arbitrary Message Bridges and the applications that use them. It allows applications to send messages across chains in a trustless manner. The protocol is designed to be chain-agnostic, meaning that it can be used with any blockchain compatible with EVM.

To use the protocol, an app must register itself in the escrow by setting the address of a remote escrow contract on a destination chain. This should be done on both the source and destination chains. The protocol is designed to be trustless and permissionless, meaning that the escrow contract does not restrict messages or destinations.

When an app sends a message, it is first sent to the escrow contract on the source chain. The escrow contract then emits an event with the message data, which is picked up by a relayer. The relayer then sends the message to the escrow contract on the destination chain. The destination escrow contract then calls the app's counterpart contract on the destination chain, which processes the message and generates the acknowledgment. The escrow on the destination chain then wraps the acknowledgment in a packet with additional metadata and emits an event. Another relayer picks up the acknowledgment and sends it to the escrow on the source chain. The source escrow then calls the source app contract on the source chain, which processes the acknowledgment, and pays the relayers for their work.

To send a message, an app must incentivize both the source and destination relayers by depositing a reward in the escrow contract on the source chain. This reward is calculated in the source chain’s native token and is split between the source and destination relayers. The reward is calculated based on the gas limits for the message and the acknowledgment, as well as the gas price. Additionally, the app pays the cost of sending the message from a chain.

Messages can include the deadline by which the message must be delivered. If the message is not delivered by the deadline, the packet is considered expired. In this case, a timeout packet is generated on the destination chain and relayed back to the source chain. In the case of a timeout, the source app contract is called with the timeout message, and the source relayer is paid for their work. The destination relayer is not paid in the case of a timeout.

Additionally, apps can specify a so-called time delta, which is the time difference between message delivery and acknowledgment. The time delta is the perfect time the app expects the acknowledgment to be delivered. The escrow protocol uses the time delta to distribute the reward between the source and destination relayers. If the acknowledgment is delivered before the time delta, the destination relayer is paid more than the source relayer. If the acknowledgment is delivered after the time delta, the source relayer is paid more than the destination relayer. If the time between message delivery and acknowledgment exceeds the time equal to twice the time delta, the destination relayer is not paid, and all the reward is paid to the source relayer.

## Contracts

Contracts we find important for better understanding are described in the following section.

#### IncentivizedMessageEscrow

The IncentivizedMessageEscrow contract is the main contract of the protocol. It is used as a middleware between Arbitrary Message Bridges (AMB) and applications. The contract is marked abstract, and its final implementation is expected to be adjusted to the specific AMB. The two virtual functions, `_sendPacket` and `_verifyPacket`, are two main functions responsible for the correct publishing and authenticity verification of messages for a specific AMB. The contract is not upgradeable and not pausable. It is also permissionless.

#### IncentivizedWormholeEscrow

The IncentivizedWormholeEscrow contract is a specific implementation of the IncentivizedMessageEscrow contract for the Wormhole AMB. The contract is not upgradeable and not pausable.

#### IncentivizedPolymerEscrow

The IncentivizedPolymerEscrow contract is a specific implementation of the IncentivizedMessageEscrow contract to use it with the Inter-Blockchain Communication (IBC) Protocol. The implementation differs from the IncentivizedMessageEscrow contract in that it disables functions like `processPacket` and `timeoutMessage` and uses the `onRecvUniversalPacket`, `onUniversalAcknowledgement` and `onTimeoutUniversalPacket` functions provided by the IBC Protocol. The message verification in this contract is done by the IBC Relayers.

### Actors

This part describes the actors of the system, their roles, and permissions.

#### App

Apps create messages, send them across chains and process acknowledgments. They set the incentives for relayers and pay for the message delivery.

#### Relayer

Relayers listen for events emitted by the escrow contracts, pick up messages, and deliver them to the destination chain. They are incentivized by the apps.

#### Escrow

The escrow contracts are specific for each AMB. They act as a middleware between the AMB and the apps. They are responsible for the correct publishing and authenticity verification of messages.

#### User

Users interact with the apps and are not directly involved in the message delivery process. However, any external user can create a timeout packet if the deadline passes. Also, they can increase the reward for the relayers and re-execute the acknowledgment packets on the source app if the execution fails.

## 5.2. Trust Model

Don’t trust, verify.

The protocol is designed to be trustless and permissionless. The escrow contracts do not restrict messages or destinations. The protocol relies on the incentives set by the apps to ensure that the relayers deliver the messages.

Apps must trust relayers to deliver the messages. The system uses 1/N security assumption, meaning that the system is secure as long as at least one of the N relayers is honest, which depends on the number of relayers. Relayers are a part of an AMB protocol.

Relayers must verify the messages and acknowledgments they deliver. Any application can register a route to a destination chain, and the destination escrow can be set to any address. This means that the relayers cannot trust all messages from the escrow, and they must either maintain a whitelist of legitimate escrow contracts or verify the deployed escrow code. Additionally, relayers must verify if the time delta set in the message is within reasonable ranges, otherwise, they risk not being paid (see W3).

## C1: Fake escrow can craft ACK packets with any `messageIdentifier` and withdraw all bounties

*Critical severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | High |
| Target: | IncentivizedMessageEscrow.sol | Type: | Data Validation |

### Description

The `processPacket` function serves as the main entry point of messages to IncentivizedMessageEscrow. First, the `_verifyPacket` function verifies the integrity of the received message and extracts the sender information. If an ACK message enters the contract, the `_handleAck` function is called internally. This function verifies if the message is from the expected escrow address on the destination chain, where the destination chain is the chain where the ACK message is generated. After this verification passes, the correct bounty structure is fetched from the storage. The search key is a message ID. However, 32 bytes of the message ID are taken from the raw message and are not validated. Moreover, the fee recipient address for relaying the message on the destination chain is also taken from the same raw message. If an attacker controls the escrow contract on the destination chain, they can craft ACK messages with IDs of packets that have not yet been acknowledged, drain all stored bounties and cause the Denial of Service for the message.

### Exploit Scenario

Consider two chains: `A` and `B`. On both chains, escrow contracts `EscrowA` and `EscrowB` exist, respectively. On the chain `A`, an AppA application is deployed to communicate with `AppB` through the escrows. The attack scenario is the following:

1. `AppA` sends a message to `AppB` through `EscrowA`. `EscrowA` takes a bounty for the message transfer and emits an event for the relayers with the generated message ID, say `MSGID1`, recipient information, and the payload. At this step, the fees are paid by `AppA` and stored in the escrow contract.
2. The attacker creates and deploys a fake escrow on chain `B`, `EvilEscrowB`. This escrow behaves almost identically to `EscrowB`; however, it can create ACK messages with arbitrary message IDs and the fee recipient address set by the attacker.
3. The attacker deploys two simple apps on `A` and `B`, `EvilAppA` and `EvilAppB`. `EvilAppA` registers itself on `EscrowA` using `setRemoteImplementation` setting the remote escrow on `B` to `EvilEscrowB`. A valid transmission route is created: `EvilAppA <> EscrowA <> EvilEscrowB <> EvilAppB`. The implementation of the apps does not matter; the only requirement is the implementation of the `ICrossChainReceiver` interface.
4. The attacker reads the newly created message ID `MSGID1` from step (1) and passes it to `EvilEscrowB`.
5. `EvilEscrowB` generates an ACK message on `B`, setting the sender of the ACK as `EvilAppB`, the message ID to `MSGID1` set by the attacker at step (4), the fee recipient, and the refund addresses to the attacker address on the chain `A`, and the recipient of the ACK message to `AppA`.
6. A relayer, which can be either controlled by the attacker or anyone else, relays the message from `B` to `A`, and the payload enters the `EscrowA` contract. The `_verifyPacket` function validates that the message from `B` is legitimate.
7. The `_handleAck` function verifies if the message to `EvilAppA` came from the expected escrow contract. Since the escrow on `B` was set to `EvilEscrowB` and `EvilEscrowB` is the sender of the ACK message, this verification is

successful.

8. The message ID `MSGID1` is read from the obtained payload, and the `IncentiveDescription` structure is read and deleted from the `_bounty` mapping.
9. The refund address and the destination relayer address are taken from the payload, too.
10. All the fees stored in the contract in step (1) are transferred to the attacker.
11. The Attacker repeats the process until all funds are drained for all messages not yet acknowledged.

### Recommendation

Verify if the message ID obtained in the payload refers to the apps that initiated the message transmissions. Make sure you thoroughly validate the payload.

### Fix 1.1

The `_bounty` mapping's type was changed from `mapping(bytes32 => IncentiveDescription)` to `mapping(address fromApplication => mapping(bytes32 destChain => mapping(bytes32 messageIdentifier => IncentiveDescription)))`. This modification allows for connecting every message ID to the sender application and the destination chain, resulting in the strict assignment of the bounty to only one route. All the functions using this mapping were updated to reflect the new structure. The issue was fixed in the PR#41, commit `d50ca3a` [1].

Go back to Findings Summary

## M1: Fee recipient addresses are not validated against the zero address

*Medium severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | Low |
| Target: | IncentivizedMessageEscrow.sol | Type: | Data Validation |

### Description

The constructor of IncentivizedMessageEscrow sets the immutable variable `SEND_LOST_GAS_TO` to a provided one in the argument. There are no validations of the correctness of the provided argument. If the provided argument is the zero address, bad consequences may exist. The first problem is that the contract transfers the native token to this address. In the case of the zero address, the tokens will be lost for messages with fee distribution problems. Secondly, this will require redeployment, and if the address should be the same on multiple EVM chains, relayers need to keep track of multiple addresses and ensure the incorrect one is never used.

The same issue is within the `processPacket()` function where the `feeRecipient` variable is used without validations and is directly passed to the `_payoutIncentive` function. If `feeRecipient` is set to `bytes32(0)`, the tokens sent to this address will be lost.

### Exploit Scenario

Assume there are three chains: `A`, `B` and `C`; they all support the `CREATE2` EVM opcode. On chains `A` and `B`, there are two escrow deployments, `EscrowA` and `EscrowB`. Both escrows are deployed using `CREATE2` by the same deployer, and with the same bytecode and salt, so both escrows share the same EVM address. The deployer integrates `C` into the system and runs a script that should deploy `EscrowC`. However, there is a bug in the deployment script, and accidentally, `EscrowC` is deployed with `SEND_LOST_GAS_TO` set to the zero address. Relayers start relaying messages from and to `C`, and because some packets have incorrectly set fee recipients, some native tokens are sent to `SEND_LOST_GAS_TO`. In the case of `EscrowC`, tokens are burned. The team noticed the problem and redeployed the escrow contract to the correct address. Now, the team has to notify relayers that `EscrowC` located on chain `C` with the same address as `EscrowA` and `EscrowB` is malfunctioning and should not be used.

## Recommendation

Add the zero-address validation to the constructor:

```
constructor(address sendLostGasTo) {
    if (sendLostGasTo == address(0)) revert SendLostGasIsZero();
    SEND_LOST_GAS_TO = sendLostGasTo;
}
```

Also, add a check to the `processPacket` function:

```
if (feeRecipient == bytes32(0)) revert FeeRecipientIsZero();
```

## Fix 1.1

In the constructor, a new check is added:

```
if (sendLostGasTo == address(0)) revert SendLostGasToIsZero();
```

In the `processPacket` function, a new check is added:

```
if (feeRecipient == bytes32(0)) revert FeeRecipientIsZero();
```

The issue was fixed in the PR#43, commit `2fbcf02` [2].

Go back to Findings Summary

## M2: Insufficient validation of a disabled route may lead to the locked Ether

*Medium severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | Low |
| Target: | IncentivizedMessageEscrow.sol | Type: | Data Validation |

### Description

The `setRemoteImplementation` function of the [IncentivizedMessageEscrow](IncentivizedMessageEscrow) contract allows for setting a remote implementation address to any `bytes` value. The function documentation states that the route may be turned off by setting the `hex"00"` value for `implementation`. However, this value is not implemented and only serves as an example. Because turning off a route is not standardized, an app can still send a message to a disabled remote chain to a non-existing address. Because this address cannot act as an escrow, the ACK packet will never be generated, and if the message deadline is zero, the bounty set on the source chain will never be paid out. The contract does not have any way to recover such undeliverable messages, and the bounty paid to the contract will become locked in the contract.

### Exploit Scenario

Assume there are two chains: `A` and `B`. On `A`, there is an app `AppA` and the escrow `EscrowA`. `AppA` calls `EscrowA.setRemoteImplementation(keccak256("B"), hex"00")`. At this point, `AppA` should not be able to send messages to chain `B`. However, `AppA` can still call `submitMessage()` with `destinationIdentifier = keccak256("B")`. The contract has no validations that would prevent `AppA` from sending the message to a disabled chain. A destination relayer cannot deliver the message to `B`. The fees paid by `AppA` are locked in `escrowA` and cannot be recovered.

### Recommendation

Consider standardizing the way to turn off the remote chain. For example, a `disableRoute(bytes32 destinationIdentifier)` function can be implemented like following pseudo-code:

```
function disableRoute(bytes32 destinationIdentifier) external {
    setRemoteImplementation(destinationIdentifier, hex"00");
}
```

Furthermore, the line in `submitMessage()`

```
if (destinationImplementation.length == 0) revert NoImplementationAddressSet();
```

can be extended to

```
if (destinationImplementation.length == 0) revert NoImplementationAddressSet();
if (destinationImplementation.length == 1 && destinationImplementation[0] ==
0x00) revert RemoteDisabled();
```

### Fix 1.1

A new error `RouteDisabled` is added to `IMessageEscrowErrors` with a new constant `bytes1 constant DISABLE_ROUTE_IMPLEMENTATION = 0x00` in IncentivizedMessageEscrow. The `setRemoteImplementation` function's documentation is updated to reflect the new standard for disabling a route. The `submitMessage` function is updated to check if the destination implementation is disabled and revert with `RouteDisabled` if it is:

```
if (destinationImplementation.length == 1 && destinationImplementation[0] ==
DISABLE_ROUTE_IMPLEMENTATION) revert RouteDisabled();
```

The issue was fixed in the PR#48, commit `cc44ec2` [3].

Go back to Findings Summary

## M3: `MessageDelivered` event is used for both successful and failed calls

*Medium severity issue*

| | | | |
|---|---|---|---|
| Impact: | Medium | Likelihood: | Medium |
| Target: | IncentivizedMessageEscrow.sol | Type: | Logging |

### Description

The message identifier and the payload in `_handleMessage` are taken from the raw message received by the destination relayer. The function verifies if the message came from the authorized source and if the deadline has not passed yet. In all the cases, an acknowledgment packet is crafted, and the `MessageDelivered(bytes32 indexed messageIdentifier)` is emitted. The event includes no additional information about what happened in the contract, and off-chain components can't distinguish successful message deliveries from unsuccessful ones. Moreover, if the same or a malformed packet is sent several times from multiple senders, accidentally or on purpose, the escrow contract will emit the same `MessageDelivered` event multiple times, and the off-chain indexing and filtering will become complicated. The worst-case scenario is the denial of service for a message because of deceived relayers that collect the information of the message delivery status and do not relay already delivered messages.

### Recommendation

Update the emitted event with more information about the party that sent the message to allow relayers to filter out faulty deliveries.

## Fix 1.1

Multiple events were updated with more information about the party that sent the message. Events `BountyPlaced`, `MessageDelivered`, `MessageAcked`, `TimeoutInitiated`, `MessageTimedOut` and `BountyClaimed` now include the information about the source or destination implementation and the chain identifier. For instance, the `MessageDelivered` event described above now includes the information about the source escrow implementation address that sent the message and the source chain identifier. The relayers are now protected against invalidating message deliveries for valid messages in case invalid senders send an unauthorized message to the escrow contract. The issue was fixed in the PR#42, commit `c490e14` [4].

Go back to Findings Summary

## L1: Large messages may not be delivered due to different block gas limits on different chains

*Low severity issue*

| | | | |
|---|---|---|---|
| Impact: | Medium | Likelihood: | Low |
| Target: | IncentivizedMessageEscrow.sol | Type: | Denial of Service |

### Description

Most EVM chains have different block gas limits. If an application on the source chain, where gas limits are high enough, sends a large message to the destination chain with a lower block gas limit, the transaction will fail. The message becomes undeliverable, and the bounty paid by the source application will become locked forever in the escrow contract.

### Exploit Scenario

Assume that an application sends a message from Arbitrum to Optimism. Arbitrum has a block gas limit of 32,000,000, while Optimism has 30,000,000. The app creates a large message processed on Arbitrum, but it is too big for Optimism. The transaction on the destination chain will revert because of the block gas limit. The incentive on the source chain gets locked.

### Recommendation

There are multiple possible solutions:

1. Set a limit to the message length.
2. Implement a function that allows cancellation of packet transmission and refund fees.

3. Set an explicit large deadline for all messages.

### Fix 1.1

The issue was acknowledged with the comment:

> We acknowledge the issue, however, we don't believe there is any way to generalize a solution. We don't want to enforce arbitrary limits (how are we to know what limits apply for a combination of chains + AMBs?) The contracts are ownerless and as a result, this would be implemented as a constant. How can one say that the message size is constant over time? Recommendation 2 is not possible without significantly increasing the gas cost. (~20-30% extra). Recommendation 3 is not a solution either since timeouts would be blocked similarly to the ordinary packages.

Go back to Findings Summary

## L2: Unfair fee distribution due to floating `block.timestamp`

*Low severity issue*

| | | | |
|---|---|---|---|
| Impact: | Low | Likelihood: | Low |
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

As `block.timestamp` is floating between different chains, there may be a problem with calculating fair fees for source and destination relayers for messages with the `timeDelta` set. There is a possibility that the `block.timestamp` for the ACK message is lower than the timestamp of the message delivery on the destination chain. The execution time of the ACK message is calculated using the following code:

*Listing 1. Excerpt from IncentivizedMessageEscrow*

```
        uint64 executionTime;
        unchecked {
            // Underflow is desired in this code chuck. It ensures that the code piece continues working
            // past the time when uint64 stops working. *As long as any timedelta is less than uint64.
            executionTime = uint64(block.timestamp) - messageExecutionTimestamp;
        }
```

In this case, if `messageExecutionTimestamp` is greater than `block.timestamp`, the subtraction underflows and `executionTime` becomes a large unsigned value, much larger than two time deltas. In this case, the destination relayer fee is zero, and the source relayer receives everything.

### Exploit Scenario

The following scenario can happen accidentally or on purpose if the source relayer controls the L2 sequencer and can set block timestamps.

Assume that a message is sent from L2 to L1. On L1, it is confirmed, and an ack message is generated. The time delta is set to be `d`, a positive number of seconds.

1. The source relayer relays the ACK message from the destination to the source chain, i.e. from L1 to L2. Assume the ACK packet contains the execution timestamp equal to `x`.
2. A new L2 block is created, and the sequencer sets `block.timestamp` to a value that happens to be lower than `x`, let `y < x`.
3. The escrow contract calculates the execution time, which underflows and is set to a large value, i.e., `2**64 - (x-y) >> 2 * d`.
4. The destination relayer gets zero fees, and the source relayer acquires `actualFee`, or the sum of the destination and source fees.

### Recommendation

Make sure that the message timestamp is in the past or assume that if it's in the future, the time elapsed from the message delivery is small.

### Fix 1.1

The `_payoutIncentive` function was updated to check if the `executionTime` variable is unrealistically large. If it is, the function sets the `executionTime` to zero. The upper boundary was chosen to be 32768 days since that is the nearest value close to the `uint32` limit or 49710 days. The check code is shown below:

```
if (executionTime > 32768 days) executionTime = 0;
```

The issue was fixed in the PR#46, commit `16827be` [5].

Go back to Findings Summary

## L3: Usage of `send` and `transfer` can make the escrow unusable for smart-contract relayers

*Low severity issue*

| Impact: | Low | Likelihood: | Low |
|---|---|---|---|
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

In IncentivizedMessageEscrow, the `_payoutIncentive` function distributes the fees between source and destination relayers and the refund address. The implementation uses `send` and `transfer` functions to send the required fees to the recipients. These functions transfer the required amount of the native token, limiting the gas to 2300. While this does not influence transfers to EOAs, this gas limitation may create problems for relayers or refund addresses pointing to smart contracts. If the recipient contract's `receive()` function has some additional logic, or if the relayer is called through a proxy contract, the token transfer will fail because of insufficient gas. Furthermore, gas prices for certain opcodes may change in the future, and transfers working today may fail after certain network upgrades.

*Listing 2. Excerpt from IncentivizedMessageEscrow*

```
        if(!payable(refundGasTo).send(refund)) {
            payable(SEND_LOST_GAS_TO).transfer(refund);  // If we don't send the gas somewhere, the gas is lost forever.
        }

        // If both the destination relayer and source relayer are the same then we don't have to figure out which fraction goes to who. For timeouts, logic should end here.
        if (destinationFeeRecipient == sourceFeeRecipient) {
            payable(sourceFeeRecipient).transfer(actualFee);  // If this
```

```
reverts, then the relayer that is executing this tx provided a bad input.
            return (gasSpentOnSource, deliveryFee, ackFee);
        }

        // If targetDelta is 0, then distribute exactly the rewards.
        if (targetDelta == 0) {
            // ".send" is used to ensure this doesn't revert. ".transfer"
could revert and block the ack from ever being delivered.
            if(!payable(destinationFeeRecipient).send(deliveryFee)) {  // If
this returns false, it implies that the transfer failed.
                // The result is that this contract still has deliveryFee.
As a result, send it somewhere else.
                payable(SEND_LOST_GAS_TO).transfer(deliveryFee);  // If we
don't send the gas somewhere, the gas is lost forever.
            }
            payable(sourceFeeRecipient).transfer(ackFee);  // If this
reverts, then the relayer that is executing this tx provided a bad input.
            return (gasSpentOnSource, deliveryFee, ackFee);
        }
```

## Exploit Scenario

There are three scenarios:

1. The `refundGasTo` is a smart contract with some additional logic, for which 2300 gas is insufficient. In this case, `send` fails, and the refund is transferred to the `SEND_LOST_GAS_TO` address. The execution continues, yet the value should be transferred manually to the refund address.
2. The `sourceFeeRecipient` is a smart contract with some additional logic, for which 2300 gas is insufficient. In this case, the transaction fails because `transfer` is used. The ACK message is not delivered, and the relayer is not paid for the gas spent. Another EOA or relayer with a simpler receive logic can deliver the ACK message.
3. The `destinationFeeRecipient` is a smart contract with some additional logic, for which 2300 gas is insufficient. This case is similar to the first one. The gas message is delivered, but the gas must be transferred manually to the destination relayer later.

### Recommendation

Consider using `call()` instead of `transfer()` and `send()`:

```
(bool transferSuccess,) = feeRecipient.call{value: value}("");
if (!transferSuccess) {
    (bool lostTransferSuccess,) = lostGasRecipient.call{value: value}("");
    require(lostTransferSuccess, "Transfer failed.");
}
```

There may also be a need to limit the gas for the call to `refundGasTo` and `destinationFeeRecipient` because they can consume all 63/64 allocated gas, and the remaining 1/64 may not be enough to finish the execution. One can either set a hard gas value and state this explicitly in the documentation or a fraction of the remaining gas.

The other option is to implement a pull mechanism, where the recipient withdraws the funds from the escrow contract. This way, all the gas-related problems are shifted to the recipient. However, the pull mechanism must be thought through carefully to support all possible AMBs architectures.

### Fix 1.1

The issue was acknowledged with the comment:

> We do not believe there is a solution that does not come with tradeoffs.
>
> - The current solution is incompatible with certain chains, certain relayer addresses, and certain refund addresses.
> - Using `call`` + adding a larger amount of gas forwarding is not a guaranteed solution. Whatever gas assumptions we make may change and this solution won't be any better than the

> current. Furthermore, depending on how large the gas stipend is, it can be used for griefing and add variability to the cost of delivering acks. (make evaluation of bounties more difficult)
>
> - Using a pull scheme makes the integration into external relayers more difficult as they now have to understand they need to pull assets. This adds UX issues where we need to inform the users that they need to pull refunds.
>
> Based on all of these observations, we have decided not to change the logic.

Go back to Findings Summary

## W1: Usage of `solc` optimizer

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | `**/*` | Type: | Compiler configuration |

### Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

### Exploit scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

### Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

### Fix 1.1

The issue was acknowledged with the comment:

> We need the Solidity optimizer to fix stack issues.

Go back to Findings Summary

## W2: `block.timestamp` can be different on different chains

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

The IncentivizedMessageEscrow contract relies on `block.timestamp` to calculate the deadlines and, in case time deltas are used, the fees for source and destination relayers. Moreover, timestamps on layer-2 blockchains are floating. For example, on Arbitrum, timestamps of L2 blocks are set by a sequencer within a floating window `[-24 hours, 1 hour]` from the actual timestamp to accommodate possible delays in posting the transaction batch onto the parent chain. This behavior and inconsistencies can cause the escrow contract to create invalid packages, especially with a small deadline. One way to use timestamps to increase the fee for a source relayer is described in L2.

### Recommendation

Time synchronization is a difficult task. One way to mitigate the influence of discrepancies between timestamps on source and destination chains is by describing these limitations in the developers' documentation to warn against using deadlines that are too short. If there is enough time for the message to propagate to the destination chain and for the acknowledgment to go back, if the time delta is adequate for a specific pair of source and target chains, the problem is not that pronounced.

### Fix 1.1

The team claimed that the issue was already known to them. Additionally, they mentioned that

> The fix from a relayer / application perspective is setting `targetDelta` to 0.

As a result, the team decided to acknowledge the issue.

Go back to Findings Summary

## W3: Too small or too large time deltas make the fee distribution unfair

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

The distribution of the fees between source and destination depends on the time delta. For messages that enable this functionality, if the ACK packet is delivered before the time delta passes from the time of message delivery on the destination chain, fees are proportionally reduced for the source relayer and increased for the destination relayer. The same principle applies if the time difference between the ACK message and message delivery exceeds the time delta. In this case, fees are proportionally reduced for the destination and given to the source relayer.

While this design incentivizes relayers to deliver messages precisely on time, these fee distribution rules can be misused. There are two cases:

1. If the party who can relay messages from the source to the destination chain creates an incentive with a large time delta and relays the message, they can wait for others to relay the ACK message back to the source chain. In this case, because of the significant difference between the time delta and the execution time, the destination relayer gets all the fees. The source relayer gets nothing or almost nothing.
2. The opposite situation is for a party who can relay ACK messages from the destination to the source chain. In this case, it sets a small time delta and lets others relay their message while handling the ACK themselves. Then, the source relayer gets all the fees, and the destination relayer gets

nothing.

### Recommendation

Ensure relayers know about this behavior and recommend adequate time delta ranges in the documentation so that relayers can either ignore malicious messages or make sure that one relayer can relay both the message and the acknowledgment.

### Fix 1.1

The team acknowledged the issue with a link to the documentation that explains the issue and suggests that relayers should be aware of this behavior.

Go back to Findings Summary

## W4: Setting insufficient gas for a call will lead to undelivered messages and locked assets

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

Message creation includes setting maximum gas limits for source and destination chains and gas prices. While users can increase the gas price, the amount of gas for calls is fixed. If gas values are insufficient for a call on the destination chain or for delivering the acknowledgment, there is a risk of the message not being executed.

### Recommendation

Consider adding a function that increases `maxGasAck` and `maxGasDelivery`.

### Fix 1.1

The team acknowledged the issue with the comment:

> It is not possible to change `maxGasDelivery` once set since it is part of the cross-chain message. If a pathway was created to change it by emitting a new message, relayers could ignore the new message (which would get proof later than the original message). Changing `maxGasAck` could act as a DoS vector where anyone could increase the maxGasAck (using custom application logic) to deny delivery of messages. This is not intended. The message failing to execute on the destination is not an issue for applications aware of the risk. Furthermore, acks

> can always be replayed so failure on the source can be circumvented. If applications want to ensure that messages are delivered with enough gas, estimate the gas that will be used at the destination and add a significant and proper margin. Unspent gas will be refunded.

Go back to Findings Summary

## W5: From applications are not validated for being a smart contract

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Data Validation |

### Description

The `_handleAck` and `_handleTimeout` functions use a low-level assembly `call` function. However, the target address (`fromApplication`) is never validated if it is a smart contract. In Solidity, calling an address without any code is always successful.

### Recommendation

Ensure that addresses being called have code deployed.

### Fix 1.1

The team acknowledged the issue with the following comment:

> Nothing prohibits an EOA from submitting messages nor should it be prohibited. The fact that they can't do anything with the ack is if nothing else intended. If they wanted to use the ack, they would make the call from an application (Instead of calling `submitMessage` directly deploy a contract that calls `submitMessage` with relevant logic for handling the ack.) The fact that there is no code size check is intended as otherwise it could deny bounties from being claimed.

Go back to Findings Summary

## W6: Paying the maximum gas fee for timeouts may incentivize relayers not to deliver messages

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

An app sending a message sets fees it will pay to destination and source relayers. If the message is successfully delivered, the spent gas is calculated based on the actual gas spending. However, if the message is delivered late, the message times out, and the gas spent on the destination chain is set to the maximum allowance, even though almost no gas was paid on the destination chain. If there are a few relayers in the system, it may incentivize them not to deliver messages and only deliver timeouts to get higher rewards.

### Recommendation

Make sure there are enough relayers available in the system to ensure the probability of higher rewards for only delivering timeouts is lower than the probability of losing the yield for fair deliveries. In this case, the system will be operational, and messages will be delivered on time.

### Fix 1.1

The issue was acknowledged with the following comment:

> We are aware. Considering that the security for this issue is 1/N, we believe competition will incentivize relayers to relay packages before timeout. When timeouts aren't executed before their deadline, it must have been because the incentive

> wasn't good enough. As a result, the incentive may be slightly higher when the message timed out which hopefully should get it relayed.

[Go back to Findings Summary]

## W7: True and logged to the event gas spent values are different

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | IncentivizedMessageEscrow.sol | Type: | Logging |

### Description

In the case of the timeout messages, the gas spent on the destination chain is set to the maximum allowed value, and the whole bounty is paid to the source relayer based on this maximum spending. However, the emitted event `BountyClaimed` has zero value for gas spent on the destination, which is inconsistent with the actual spending.

*Listing 3. Excerpt from IncentivizedMessageEscrow*

```
        (uint256 gasSpentOnSource, uint256 deliveryFee, uint256 ackFee) =
_payoutIncentive(
            gasLimit,
            maxGasDelivery, // We set gas spent on destination as the entire
allowance.
            maxGasDelivery,
            priceOfDeliveryGas,
            maxGasAck,
            priceOfAckGas,
            refundGasTo,
            address(uint160(uint256(feeRecipient))),
            address(uint160(uint256(feeRecipient))),
            0, // Disable target delta, since there is only 1 relayer.
            0
        );

        emit MessageTimedOut(messageIdentifier);
        emit BountyClaimed(
            messageIdentifier,
            0,  // No Gas spent on destiantion chain.
            uint64(gasSpentOnSource),
            uint128(deliveryFee),
            uint128(ackFee)
        );
```

### Recommendation

Consider changing the gas value in the event from zero to `maxGasDelivery`.

### Fix 1.1

The `_handleTimeout` function was updated to emit the correct gas value in the `BountyClaimed` event, or `uint64(maxGasDelivery)`. The issue was fixed in the PR#45, commit `3c3bf30` [6].

Go back to Findings Summary

## W8: Relayers are not protected against a malicious escrow on the destination chain

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Logic |

### Description

The IncentivizedMessageEscrow contract allows applications to set a remote escrow address to any value. Because the source escrow is trusted, relayers may relay the message to the destination chain and unintentionally call a malicious contract. While this permissionless design makes the escrow universal and provides more flexibility, relayers must manage their whitelists of valid escrow contracts to avoid interacting with malicious code.

### Recommendation

State the permissionless design explicitly in the documentation. Ensure new relayers understand the risks associated with delivering messages without additional validation.

### Fix 1.1

The issue was acknowledged with the following response:

> We have designed the contract to be verifiable purely based on address via `create2`. If you know the AMB address, the Generalised Incentives contract will be deployed to a predetermined address. It is still assumed that relayers keep track of escrow addresses to monitor events from, as any external contract may send events with the same topic 0 but

> without any logic to handle incentives. Only tracking messages from trusted escrow implementations also simplifies storage as message identifiers are promised to not collide.

Go back to Findings Summary

## W9: A compiler bug may create dirty storage bytes

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Compiler Bugs |

### Description

In the `setRemoteImplementation()` function, the line `implementationAddress[msg.sender][destinationIdentifier] = implementation` copies bytes from `calldata` to `storage`. The minimum Solidity version of the contract is set to 0.8.13; this version is prone to a bug that may result in dirty storage values while copying bytes from `calldata` to `storage`.

### Recommendation

Consider using the latest Solidity version.

### Fix 1.1

The minimum Solidity version was changed to 0.8.22 in all contracts. The issue was fixed in the PR#47, commit `b4e27b2` [7].

Go back to Findings Summary

## I1: Unused declarations

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | IncentivizedPolymerEscrow.sol | Type: | Code Style |

### Description

In IncentivizedPolymerEscrow, the `NotEnoughGasProvidedForVerification` error and the `_TIMEOUT_AFTER_BLOCK` constant are never used.

### Recommendation

Consider removing unused declarations.

### Fix 1.1

The team claimed that the IncentivizedPolymerEscrow contract was outdated and a new version was created that was out of the scope of the audit. The issue was therefore acknowledged with the comment:

> Fixed in the newest version of the Polymer Escrow.

Go back to Findings Summary

## I2: Improve protocol documentation

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | IncentivizedMessageEscrow.sol | Type: | Documentation |

### Description

The NatSpec documentation of several functions is either missing or incomplete. Good documentation shows the maturity of the project and improves the user experience.

- `IncentivizedMessageEscrow::_verifyPacket`: Missing parameters documentation.
- `IncentivizedMessageEscrow::proofValidPeriod`: Missing documentation.
- `IncentivizedMessageEscrow::bounty`: Missing documentation.
- `IncentivizedMessageEscrow::messageDelivered`: Missing parameters documentation.
- `IncentivizedMessageEscrow::setRemoteImplementation`: The documentation for the `implementation` format and parameters documentation is missing.
- `IncentivizedMessageEscrow::increaseBounty`: Missing parameters documentation.
- `IncentivizedMessageEscrow::submitMessage`: Missing `incentive` parameter documentation.
- `IncentivizedMessageEscrow::recoverAck`: Missing parameters documentation.
- `IncentivizedMessageEscrow::reemitAckMessage`: Missing parameters documentation.

- `IncentivizedMessageEscrow::timeoutMessage`: The order of `sourceIdentifier` and `implementationIdentifier` is swapped.

#### Recommendation

Consider improving the contract documentation and adding missing information.

#### Fix 1.1

The missing or incomplete documentation for the IncentivizedMessageEscrow contract was updated. The issue was fixed in the PR#44, commit `9846038` [8].

Go back to Findings Summary

## I3: Use maximum line length

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | `**/*` | Type: | Code Style |

### Description

The code in the project does not adhere to a maximum line length, resulting in code that is difficult to read and comprehend. This violation of coding standards reduces the readability and maintainability of the code, potentially leading to errors and inefficiencies.

### Recommendation

Consider refactoring the codebase to maintain a maximum line length, typically around 80-120 characters per line.

### Fix 1.1

The issue was acknowledged with the comment:

> We will consider this for future smart contracts.

Go back to Findings Summary

[1] full commit hash: `d50ca3a08b234f4fedb23063b41ffd05b65b52c3`

[2] full commit hash: `2fbcf02fc6ff363864301923d38274b8bc393c24`

[3] full commit hash: `cc44ec2795e783ba00e9e9eb3a22e5aea8e182ce`

[4] full commit hash: `c490e149b1c0116048f41cecf27af4503f98abb2`

[5] full commit hash: `16827beb92e8652df642c263ffe2f2c78303f4a7`

[6] full commit hash: `3c3bf300e3ff5e0ac30d197b715c7b1b13f722aa`

[7] full commit hash: `b4e27b2ef6e6c2754b64016517d470a0fb1b8276`

[8] full commit hash: `9846038457621fce58fb1303e64d5eb614579658`

# 6. Report revision 1.1

## 6.1. System Overview

There were no significant changes to the system since the last review. All the issues identified in the previous review have been addressed.

### Contracts

No contracts were added or removed since the last review.

### Actors

No actors were added or removed since the last review.

## 6.2. Trust Model

The trust model has not changed since the last review.

# Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Catalyst: Generalised Incentives, 06.05.2024.

# Appendix B: Glossary of terms

The following terms might be used throughout the document:

**Superclass/Ancestor of C**

A contract that C inherits/derives from.

**Subclass/Child of C**

A contract that inherits/derives from C.

**Syntactic contract**

A Solidity contract. May have an inheritance chain, and may be deployed.

**Deployed contract**

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

**Init/initialization function**

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

**External entrypoint**

A `public` or `external` function.

**Public/Publicly-accessible function/entrypoint**

An `external` or `public` function that can be successfully executed by any network account.

**Mutating function**

A non-`view` and non-`pure` function.

# Appendix C: Fee calculation fuzzing

The contract uses complex calculations to determine the fee for the source and destination relayer in the case when the time delta is set for a message.

*Listing 4. Excerpt from [IncentivizedMessageEscrow]()*

```
        uint64 executionTime;
        unchecked {
            // Underflow is desired in this code chuck. It ensures that the code piece continues working
            // past the time when uint64 stops working. *As long as any timedelta is less than uint64.
            executionTime = uint64(block.timestamp) - messageExecutionTimestamp;
        }
        // The incentive scheme is as follows: When executionTime = targetDelta then
        // The rewards are distributed as per the incentive spec. If the time is less, then
        // more incentives are given to the destination relayer while if the time is more,
        // then more incentives are given to the sourceRelayer.
        uint256 forDestinationRelayer = deliveryFee;
        unchecked {
            // |targetDelta - executionTime| < |2**64 + 2**64| = 2**65
            int256 timeBetweenTargetAndExecution = int256(uint256(executionTime))-int256(uint256(targetDelta));
            if (timeBetweenTargetAndExecution <= 0) {
                // Less time than target passed and the destination relayer should get a larger chunk.
                // targetDelta != 0, we checked for that.
                // max abs timeBetweenTargetAndExecution = | - targetDelta| = targetDelta => ackFee * targetDelta < actualFee * targetDelta
                //  2**127 * 2**64 = 2**191
                forDestinationRelayer += ackFee * uint256(-timeBetweenTargetAndExecution) / targetDelta;
            } else {
                // More time than target passed and the ack relayer should get a larger chunk.
                // If more time than double the target passed, the ack relayer should get everything
                if (uint256(timeBetweenTargetAndExecution) < targetDelta) {
                    // targetDelta != 0, we checked for that.
                    // max abs timeBetweenTargetAndExecution = targetDelta since we have the above check
                    // => deliveryFee * targetDelta < actualFee * targetDelta < 2**127 * 2**64 = 2**191
                    forDestinationRelayer -= deliveryFee * uint256(timeBetweenTargetAndExecution) / targetDelta;
                } else {
                    // This doesn't discourage relaying, since executionTime first begins counting once the destination call has been executed.
                    // As a result, this only encourages delivery of the ack.
                    forDestinationRelayer = 0;
                }
            }
        }
```

To fuzz this calculation, we extracted the relevant code into a separate contract `MockTestDeltaCalc.sol` with only one function with the following signature:

```
function testDelta(
    uint64 targetDelta,
    uint256 messageExecutionTimestamp,
    uint256 ackExecutionTimestamp,
    uint256 ackFee,
    uint256 deliveryFee
) external view returns(uint256 forDestinationRelayer, uint256
forSourceRelayer);
```

We created a differential test using Python and Wake. The implemented calculations part of the test is shown below:

```
def reference(
        self,
        targetDelta: int,
        messageExecutionTimestamp: int,
        ackExecutionTimestamp: int,
        ackFee: int,
        deliveryFee: int
) -> Tuple[int, int]:
    ackExecutionTimestamp = ackExecutionTimestamp
    messageExecutionTimestamp = messageExecutionTimestamp
    executionTime = ackExecutionTimestamp - messageExecutionTimestamp
    executionTimeUint64 = executionTime % 2**64
    if executionTimeUint64 == targetDelta:
        return deliveryFee, ackFee
    elif executionTimeUint64 < targetDelta:
        return (
            int(deliveryFee + ackFee * (targetDelta - executionTimeUint64) /
targetDelta),
            int(ackFee - ackFee * (targetDelta - executionTimeUint64) /
targetDelta),
        )
    else:
        if executionTimeUint64 >= targetDelta * 2:
            return (0, ackFee + deliveryFee)
        else:
            return (
                int(deliveryFee - deliveryFee * (executionTimeUint64 -
targetDelta) / targetDelta),
                int(ackFee + deliveryFee * (executionTimeUint64 - targetDelta) /
targetDelta),
            )
```

The fuzz test runs 100,000 random tests with different data to check if the values returned by the contract are close to the values returned by the reference implementation. The closeness is defined as the relative difference between values is at max 0.001%. The implementation of the test is shown below:

```
@flow()
def flow_test_time_delta(self):
    targetDelta = random_int(1, 2**64 - 1, edge_values_prob=0.2)
    messageExecutionTimestamp = random_int(1000, 2**256-1)
    ackExecutionTimestamp = messageExecutionTimestamp + random_int(-120, 120)
    ackFee = random_int(1, 2**144)
    deliveryFee = ackFee

    tx = self.mock.testDelta(
        targetDelta,
        messageExecutionTimestamp,
        ackExecutionTimestamp,
        ackFee,
        deliveryFee,
        request_type="tx"
    )
    forDestinationRelayer, forSourceRelayer = tx.return_value
    refDst, refSrc = self.reference(
        targetDelta,
        messageExecutionTimestamp,
        ackExecutionTimestamp,
        ackFee,
        deliveryFee
    )
    # The problem with exact assertions is that the contract may calculate
    # the fees with an error. If 100% of the fees should go to only one
    # relayer, the contract may be "almost" correct and give 99.999% to one
    # and 0.001% to the other. However, these 0.001% may be quite large
    # integers, and the Python code will output exactly 0. The relative diff
    # between these values will be huge (goes to infinity). This's why we
    # compare the percentages of the total fees rather than the absolute values
    # with a tolerance.
    totalFee = ackFee + deliveryFee
    dstPercentage = forDestinationRelayer / totalFee
    srcPercentage = forSourceRelayer / totalFee
    dstRefPercentage = refDst / totalFee
    srcRefPercentage = refSrc / totalFee
    try:
        # comparison with zero with relative tolerance will fail, too small
numbers
        # we compare percentages, so abs_tol=0.00001 (or 0.001%) is enough
        assert math.isclose(dstPercentage, dstRefPercentage, abs_tol=1e-5)
        assert math.isclose(srcPercentage, srcRefPercentage, abs_tol=1e-5)
    except AssertionError as e:
        print(tx.console_logs)
        raise
```

# Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain