Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Generalised Incentives

Veridise Inc.

March 21, 2024

► **Prepared For:**

Cata Labs
https://catalabs.org/

► **Prepared By:**

Bryan Tan
Nicholas Brown

► **Contact Us:** contact@veridise.com

► **Version History:**

Mar. 21. 2024	V1.01 - fixed some typos
Mar. 20. 2024	V1

Contents

1 Executive Summary

From Mar. 6, 2024 to Mar. 8, 2024, Cata Labs engaged Veridise to review the security of an extension to their Generalised Incentives protocol. The review covered a smart contract implementing standardized incentives structure for relaying messages through an Arbitrary Message Bridge (AMB). Compared to the previous version, which Veridise has audited in the past*, the new version allows for attaching deadlines to messages being sent (after which the message will not be executed) and allows relayers to be paid for notifying an application of a message timeout. Veridise conducted the assessment over 6 person-days, with 2 engineers reviewing code over 3 days on commit `4abd57c74150a`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Code assessment. The Generalised Incentives developers provided the source code of the updated version of the Generalised Incentives contracts for review. This code is original code written by the developers. It contains some documentation in the form of an extensive README and documentation comments on functions and storage variables.

The source code contains a test suite, which the Veridise auditors noted covered an extensive variety of scenarios, including both successful and unsuccessful scenarios in the protocol.

Summary of issues detected. The audit uncovered 13 issues, 1 of which is assessed to be of high severity by the Veridise auditors. Specifically, messages sent from unauthenticated source escrow contracts could modify the message delivery status of any message, causing a denial-of-service attack (V-GNI2-VUL-001). The Veridise auditors also identified 2 medium-severity issues, including a problem where message identifiers generated by a source chain escrow do not include sufficient information to uniquely identify a message in some scenarios (V-GNI2-VUL-002), and missing validation of a message or ack packet's authenticity may be combined with other vulnerabilities to create exploits (V-GNI2-VUL-003); as well as 2 low-severity issues, 7 warnings, and 1 informational finding.

Among the 13 issues, 10 issues have been acknowledged by Cata Labs, and 3 issues have been determined to be intended behavior after discussions with Cata Labs. Of the 10 acknowledged issues, Cata Labs has fixed 7 issues. This includes the 1 medium-severity issue and 1 high-severity issue. Cata Labs does not plan to fix the other 3 acknowledged issues at this time. The Veridise auditors have included in the report the 3 issues determined to be intended behavior, so that readers are aware of behavior which may at first be unexpected.

Recommendations. After completing the audit, the auditors had a few recommendations to improve the security of the codebase going forward. First, we recommend clearly documenting all assumptions made by the protocol in a centralized location, including any properties that

* The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

must be satisfied when applications configure the escrow contracts. For example, it may be helpful to write an explicit list of access control policies such as "unauthenticated senders of a message packet should receive an ack packet with a `NO_AUTHENTICATION` error code" or "an ack packet sent by an escrow address that does not match the stored remote implementation for the sender's chain should be rejected". Most of the attacks identified in this audit occur by exploiting a combination of vulnerabilities like V-GNI2-VUL-003, V-GNI2-VUL-004, V-GNI2-VUL-001, and V-GNI2-VUL-006 that involve minor flaws in the implementations of the (implicit) security policies.

Furthermore, the auditors were confused by some of the naming conventions (e.g., "source", "destination", "implementation") when reading the code. We recommend using less ambiguous terminology (e.g., "escrow address" instead of "implementation identifier") and documenting the standard terms used in the code.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Generalised Incentives	`4abd57c74150a`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 6 - Mar. 8, 2024	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	2	2	1
Low-Severity Issues	2	2	2
Warning-Severity Issues	7	4	2
Informational-Severity Issues	1	1	1
TOTAL	13	10	7

Table 2.4: Category Breakdown.

Name	Number
Data Validation	6
Logic Error	3
Maintainability	3
Denial of Service	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Generalised Incentives's smart contracts. In our audit, we sought to answer questions such as:

- Is it possible to timeout a message that has been successfully executed on the destination chain?
- Can the timeout mechanism be exploited for a denial-of-service attack or to profit in some way?
- Is there a way for someone other than a destination-to-source relayer to claim an incentive for a message, or for a relayer to claim an incentive twice for the same message?
- Is it possible for a relayer to modify a message before it is delivered and still have the message accepted?
- Does the escrow contract correctly reject packets that are sent from unknown contracts?
- Are ack and timeout packets correctly checked for authenticity when received by a source chain?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. This tool is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the the changes made since our previous audit to the `src` folder of the source code provided by the Generalised Incentives developers.

Methodology. Veridise auditors reviewed the reports of previous audits for Generalised Incentives, inspected the provided tests, and read the Generalised Incentives documentation. They then began a manual review of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Generalised Incentives developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-GNI2-VUL-001	_messageDelivered still updated for unauthentic..	High	Fixed
V-GNI2-VUL-002	Potential message identifier hash collisions	Medium	Fixed
V-GNI2-VUL-003	Message identifier not validated for message an...	Medium	Acknowledged
V-GNI2-VUL-004	reemitAckMessage() does not check validity of s...	Low	Fixed
V-GNI2-VUL-005	Deadline of 0 is not rejected by _verifyTimeout()	Low	Fixed
V-GNI2-VUL-006	Mock _proofValidPeriod inconsistent with docume	Warning	Fixed
V-GNI2-VUL-007	timeoutMessage() does not check validity of sou...	Warning	Intended Behavior
V-GNI2-VUL-008	Code Duplication between _handleAck and _hand	Warning	Acknowledged
V-GNI2-VUL-009	Unvalidated assumption that timeoutMessage() pa	Warning	Fixed
V-GNI2-VUL-010	To-application address length not validated	Warning	Acknowledged
V-GNI2-VUL-011	Ack packet sent instead of timeout packet when ..	Warning	Intended Behavior
V-GNI2-VUL-012	Timeout packet awards delivery incentives to th...	Warning	Intended Behavior
V-GNI2-VUL-013	Typos and Comment Clarifications	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-GNI2-VUL-001: _messageDelivered still updated for unauthenticated senders

Severity	High	**Commit**	4abd57c
Type	Logic Error	**Status**	Fixed
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)			
Confirmed Fix At	`https://github.com/catalystdao/GeneralisedIncentives/pull/40`		

During the audit, the developers identified a denial-of-service attack affecting `_handleMessage()` that can be used to disable any destination chain escrow. The auditors describe and analyze the attack here.

The `_handleMessage()` method uses the `_messageDelivered` mapping to track which source-to-destination packets have already already been processed by the escrow. In particular, when `_handleMessage()` is called, it will first check whether `_messageDelivered` has a nonzero entry for the message identifier contained in the packet. If there is a nonzero entry, then the method will revert as this source-to-destination packet has already been processed. Otherwise, a response packet will be generated, and the hash of the response data will be stored in `_messageDelivered`.

```
function _handleMessage(bytes32 sourceIdentifier, bytes memory
    sourceImplementationIdentifier, bytes calldata message, bytes32 feeRecipient,
    uint256 gasLimit) internal returns(bytes memory receiveAckWithContext) {
    // Ensure message is unique and can only be execyted once
    bytes32 messageIdentifier = bytes32(message[MESSAGE_IDENTIFIER_START:
    MESSAGE_IDENTIFIER_END]);

    // The 3 next lines act as a reentry guard, so this call doesn't have to be
    protected by reentry.
    // We will re-set _messageDelivered[messageIdentifier] again later as the hash of
     the ack, however, we need re-entry protection
    // so applications don't try to claim incentives multiple times. So, we set it
    now and change it later.
    bytes32 messageState = _messageDelivered[messageIdentifier];
    if (messageState != bytes32(0)) revert MessageAlreadySpent();
    _messageDelivered[messageIdentifier] = bytes32(uint256(1));

    // ... other code to handle the packet ...
    receiveAckWithContext = bytes.concat(/* ... */);
    _messageDelivered[messageIdentifier] = keccak256(receiveAckWithContext);
    // ... other code ...
```

Snippet 4.1: Snippet in `_handleMessage()` that checks the `_messageDelivered` entry.

Because `_handleMessage()` *always* sets a nonzero `_messageDelivered` entry for *any* message identifier that has not been delivered yet, an attacker can abuse this logic to perform a denial-of-service attack, such as the one described below:

1. An attacker can set up a malicious escrow contract to use the same `sourceIdentifier` as an existing escrow and frontrun messages sent by the existing escrow.

2. The malicious escrow creates and sends fake messages by copying the existing escrow's messages.
3. If a fake message sent by the malicious escrow is received by the destination escrow before the original message is delivered, the `_handleMessage()` method will mark the message identifier as "delivered" (due to the malicious escrow being an unauthenticated sender).
4. When `_handleMessage()` is called to process the original message, it will revert due to the original message having the same message identifier as the (already-delivered) fake message.

```
bytes32 expectedSourceImplementationHash = implementationAddressHash[toApplication][
    sourceIdentifier];
if (expectedSourceImplementationHash != keccak256(sourceImplementationIdentifier)) {
    acknowledgement = bytes.concat(
        NO_AUTHENTICATION,
        message[CTX0_MESSAGE_START: ]
    );
    receiveAckWithContext = bytes.concat(/* ... */);
    _messageDelivered[messageIdentifier] = keccak256(receiveAckWithContext);
    emit MessageDelivered(messageIdentifier);
    return receiveAckWithContext;
}
```

Snippet 4.2: The snippet in `_handleMessage()` that handles unauthenticated senders. This is also where the denial-of-service attack can occur. Note that despite the source explicitly being acknowledged as unauthenticated, the message is still marked as delivered. Thus, the escrow will reject any future, legitimate packet with the same message identifier.

Impact

- When combined with a frontrunning attack like in the scenario described above, an attacker can prevent legitimate messages sent by a source chain escrow from being processed by the destination chain escrow.
- An attacker can generate an ack packet with a `NO_AUTHENTICATION` error code for any message identifier and message. This may enable replay attacks when combined with V-GNI2-VUL-004, V-GNI2-VUL-002, and/or V-GNI2-VUL-003.

Recommendation

- Address V-GNI2-VUL-002 to ensure that similar messages sent by different escrow addresses, to different applications, etc. will have unique message identifiers. This will reduce the likelihood that a destination chain escrow could be exploited to perform replay attacks.
- To prevent the denial-of-service attack, consider reverting on unauthenticated messages instead of sending back a packet with an error code. An unauthenticated sender likely will only appear if the protocol is misconfigured (which is permanent due to how `setRemoteImplemention()` works), or if the sender is truly attempting to perform some malicious action.

Developer Response The developers changed `_messageDelivered` to be a nested mapping, where it must be indexed by the source identifier, the source escrow's address, and the message identifier. This would separate the message delivery status by source identifier and source escrow address, thereby reducing the likelihood of denial-of-service attacks and mitigating against replay attacks (such as described in V-GNI2-VUL-004).

4.1.2 V-GNI2-VUL-002: Potential message identifier hash collisions

Severity	Medium	**Commit**	4abd57c
Type	Denial of Service	**Status**	Fixed
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_getMessageIdentifier()		
Confirmed Fix At	`https://github.com/catalystdao/GeneralisedIncentives/pull/40`		

The `_getMessageIdentifier()` helper function is used to deterministically construct a unique message identifier for each new instance of a request-response process in the protocol. A source chain escrow will generate a message identifier when sending a source-to-destination packet so that it can track which timeout and ack packets will correspond to the responses.

In the `IncentivizedMessageEscrow`, the message identifier is computed as a keccak256 hash of several fields of a packet, such as the "from application" that originally sent the message. However, the hash computation does not involve the following information:

- The address of the escrow contract that sent the message from the source chain ("source implementation identifier" / source escrow address).
- The address of the escrow contract that should receive the message on the destination chain ("destination implementation identifier" / destination escrow address).
- The address of the application that should be invoked on the destination chain ("to application").

```
function _getMessageIdentifier(
    address messageSender,
    uint64 deadline,
    uint256 blockNumber,
    bytes32 sourceIdentifier,
    bytes32 destinationIdentifier,
    bytes memory message
) pure internal virtual returns(bytes32) {
    return keccak256(
        bytes.concat(
            bytes20(messageSender),
            bytes8(deadline),
            bytes32(blockNumber),
            sourceIdentifier,
            destinationIdentifier,
            message
        )
    );
}
```

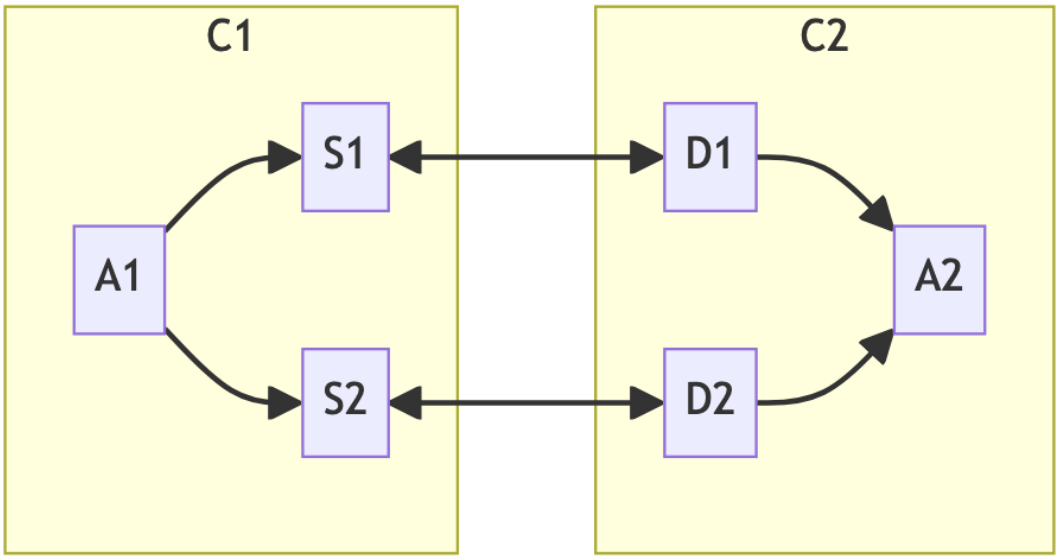
Snippet 4.3: Implementation of `_getMessageIdentifier()`.

Impact The protocol assumes that the message identifier is unique to each message; a hash collision will violate this assumption. Notably, the `_bounty` mapping on the source chain will use the message identifier to track the incentives that should be paid out on an ack or timeout packet,

and the `_messageDelivered` mapping on the destination chain will associate each message identifier to the hash of the ack packet it sent after receiving a source-to-destination packet.

Some situations where the message identifiers of two different packets can be the same include:

- Identical packets sent from different `IncentivizedMessageEscrow` contracts on the same source chain to the same destination chain will have the same message identifier. Note that each packet could be sent to different destination escrow addresses and still have the same message identifier.
  For example, consider the following configuration, where there are two chains C1 and C2. Application A1 on C1 will use escrows S1 and S2 to communicate with application A2 on C2 through escrows D1 and D2, respectively.



  In this scenario, it is possible for A1 to send the "same" packet through S1 and S2, but the message identifiers for the packet sent by S1 and the packet sent by S2 will be the same.
- Two source-to-destination packets can differ in the "to application" and "max gas" fields. This means that if an application attempts to call `submitMessage()` twice in the same block with the same destination chain and message but two different applications, then the second call will revert.

When combined with other vulnerabilities such as V-GNI2-VUL-001 or V-GNI2-VUL-004, a hash collision can cause an invalid message to be falsely accepted or a valid message to be falsely rejected. However, the auditors otherwise did not identify any concrete exploit that could occur when using hash collisions only.

Recommendation Remove the possibilities of hash collisions by ensuring that the message identifier is also computed over the missing information. This will improve the global safety of

the protocol across all deployed instances of the `IncentivizedMessageEscrow` contract.

Developer Response The developers updated `_getMessageIdentifier()` to also compute the hash over the source escrow's address.

4.1.3 V-GNI2-VUL-003: Message identifier not validated for message and ack packets

Severity	Medium	**Commit**	4abd57c
Type	Data Validation	**Status**	Acknowledged
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_handleMessage(), _handleAck()		
Confirmed Fix At	N/A		

A message identifier is used to uniquely identify an instance of a request-response process in the protocol. When a message is sent from a source chain with `submitMessage()`, the source chain escrow will generate a message identifier by hashing information related to the message and involved chains.

When a packet is processed in `_handleMessage()` (called on a destination chain) and `_handleAck()` (called on a source chain), the message identifier is read from the packet, but it is not validated to match the expected message identifier for the fields of the message.

Specifically, for `_handleMessage()`:

- The source chain is not checked to correspond to the message identifier.
- The "from application" is not checked to correspond to the message identifier.

And for `_handleAck()`:

- The destination chain is not checked to correspond to the message identifier.
- There is no validation that the the destination chain escrow that sent the ack packet is the same as the intended recipient of the original source-to-destination packet.
- The "from application" is not checked to correspond to the message identifier.

```
1 function _handleMessage(bytes32 sourceIdentifier, bytes memory
      sourceImplementationIdentifier, bytes calldata message, bytes32 feeRecipient,
      uint256 gasLimit) internal returns(bytes memory receiveAckWithContext) {
2     // Ensure message is unique and can only be execyted once
3     bytes32 messageIdentifier = bytes32(message[MESSAGE_IDENTIFIER_START:
      MESSAGE_IDENTIFIER_END]);
4     // The 3 next lines act as a reentry guard, so this call doesn't have to be
      protected by reentry.
5     // We will re-set _messageDelivered[messageIdentifier] again later as the hash of
       the ack, however, we need re-entry protection
6     // so applications don't try to claim incentives multiple times. So, we set it
      now and change it later.
7     bytes32 messageState = _messageDelivered[messageIdentifier];
8     if (messageState != bytes32(0)) revert MessageAlreadySpent();
9     _messageDelivered[messageIdentifier] = bytes32(uint256(1));
```

Snippet 4.4: Snippet where the message identifier is read in `_handleMessage()`

Impact Currently, `_handleMessage()` and `_handleAck()` rely on the underlying messaging protocol to ensure the authenticity of the contents of packets that the escrow receives (i.e., that the packet itself has not been tampered with). Since neither method validates the message

```
function _handleAck(bytes32 destinationIdentifier, bytes memory
    destinationImplementationIdentifier, bytes calldata message, bytes32 feeRecipient
    , uint256 gasLimit) internal {
    // Ensure the bounty can only be claimed once.
    bytes32 messageIdentifier = bytes32(message[MESSAGE_IDENTIFIER_START:
    MESSAGE_IDENTIFIER_END]);
```

Snippet 4.5: Snippet where the message identifier is read in `_handleAck()`

identifier against the sending escrow's chain and address, the methods themselves do not include protection against packet spoofing.

Furthermore, both methods do not validate that the sender of the packet is authorized to send the packet; they only check that the sender is *a* valid remote implementation for the chain the packet is sent from. Specifically, `_handleMessage()` does not check, for example, whether the message packet sent by an (approved) escrow `S0` was actually a packet generated by `S0` (rather than, say, an escrow `S1`). Similarly, `_handleAck()` does not check whether the ack packet was sent by an (approved) escrow `D` corresponds to the message packet that was originally sent to `D`.

Since the message identifier is not validated, this may enable attacks such as:

- A buggy source chain escrow could generate a message identifier that is inconsistent with the data in the actual packet. For example, an incorrect destination identifier may have been specified for the packet. A destination chain `IncentivizedMessageEscrow` will still accept such packets.
- If an attacker is able to modify the "from application" of an ack packet, an attacker could bypass the remote implementation check in `_handleAck()`.

```

address fromApplication = address(bytes20(message[FROM_APPLICATION_START_EVM:
    FROM_APPLICATION_END]));

// First check if the application trusts the implementation on the destination chain.
bytes32 expectedDestinationImplementationHash = implementationAddressHash[
    fromApplication][destinationIdentifier];
// Check that the application approves the source implementation
// For acks, this should always be the case except when a fradulent applications
    sends a message to this contract.
if (expectedDestinationImplementationHash != keccak256(
    destinationImplementationIdentifier)) revert InvalidImplementationAddress();
```

Snippet 4.6: Snippet in `_handleAck()` that checks the "from application" against the sender. If the attacker can change the "from application" field, they will be able to set the `expectedDestinationImplementationHash` to any desired value.

- As noted by the developers, a malicious, fake escrow can perform a denial-of-service attack on `_handleMessage()` on an arbitrary destination chain escrow, as described in V-GNI2-VUL-002.
- In `_handleAck()`, an attacker that is able to spoof destination chain details (such as by exploiting V-GNI2-VUL-004) may be able to cause the application's `receiveAck()` callback to be executed with a different destination chain than intended.

- If an application does not configure the escrows it uses correctly, it may open itself up to replay attacks that exploit V-GNI2-VUL-004 and V-GNI2-VUL-002.

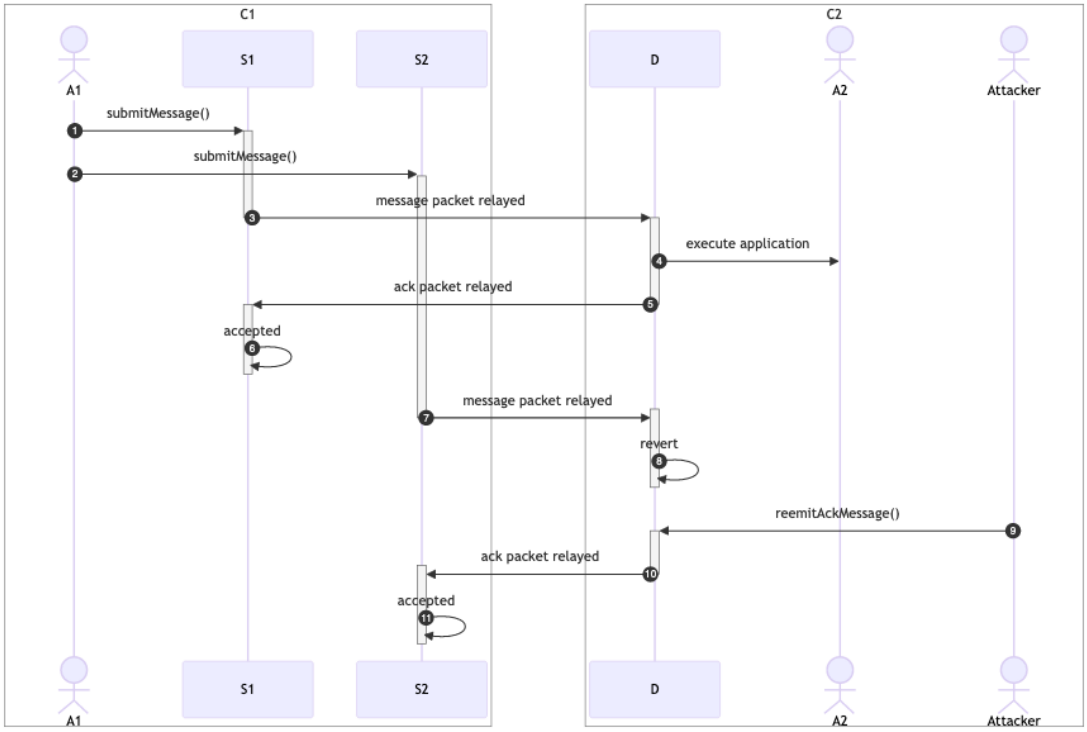


Figure 4.7: A diagram showing an example of how a replay attack can be executed against a misconfigured escrow. In this scenario, an application A1 configures escrow D on chain C2 as the implementation of both S1 and S2 on chain C1. However, application A2 on chain C2 configures S1 as the implementation of D on chain C1. If A1 uses S1 and S2 to send the same message to D, an attacker can call `reemitAckMessage()` to replay an ack packet that was previously sent to S1. This would allow the relayer of S2's message to claim incentives, even though the desired transaction was never executed on D for S2. The attack can be prevented if the message identifier computation includes S2's address and if S2 validates the message identifier against the ack packet it receives.

Recommendation

- In `_handleAck()`, validate the message identifier with respect to the message contents, the destination chain identifier and destination escrow address. This may require adding deadline and origin blocknumber fields to ack packets, or require extending the `_bounty` mapping to store these two pieces of information. Note that similar validation is already performed in `_handleTimeout()`.
- Implement the fixes in V-GNI2-VUL-002 to avoid hash collisions for message identifiers that match the contents of their messages.
- Fix V-GNI2-VUL-004 to reduce the attack surface of this issue.

Developer Response The developers acknowledge the issue but note that validation cannot be added to `_handleMessage()` for the following reason:

> We don't know what the source messageIdentifier was hashed with. We assume that keccak256 (or sha256) may not be available on all chains this is deployed to.

Furthermore, they do not intend to add validation to `_handleAck()` as that may increase gas costs significantly.

4.1.4 V-GNI2-VUL-004: reemitAckMessage() does not check validity of source chain

Severity	Low	**Commit**	4abd57c
Type	Data Validation	**Status**	Fixed
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	reemitAckMessage()		
Confirmed Fix At	`https://github.com/catalystdao/GeneralisedIncentives/pull/40`		

The `reemitAckMessage()` method can be called on a destination chain in order to re-emit a response to a previously delivered source-to-destination packet. Similar to V-GNI2-VUL-007, the `reemitAckMessage()` method does not validate that:

- The provided source-to-destination packet `message` was actually sent from the given `sourceIdentifier`.
- The `implementationIdentifier` was the same source chain escrow that the original response was sent to.

Impact The impact is largely the same as that of V-GNI2-VUL-007, except that a malicious party must call `reemitAckMessage()` instead. This would require that the message identifier has already been marked as delivered (i.e., that there was a previous successful call to `processPacket()` that supplied a packet with the same message identifier). Thus, it is likely that re-emitting an ack to a different `sourceIdentifier` (i.e., chain identifier) will fail; however, it may be possible to emit to a different `implementationIdentifier` (i.e., address of escrow) on the same source chain.

This issue may enable replay attacks when combined with other vulnerabilities such as V-GNI2-VUL-003 and V-GNI2-VUL-002. One such hypothetical scenario is described below:

1. Suppose a destination chain escrow is communicating with two different escrow contracts on the same source chain.
2. The source chain escrows use identical formats for their message identifiers.
3. The same message (and message identifier) is sent from each source chain escrow to the destination chain, but with different applications on the destination. This may be possible if an application configures the protocol incorrectly, for example.
4. One of the messages is marked as delivered due to V-GNI2-VUL-002, while the other causes a revert. An ack packet is sent back to the source chain escrow whose message was delivered, causing the `receiveAck()` callback to be executed on the application.
5. An attacker calls `reemitAckMessage()` to send an ack packet to the source chain escrow whose message caused a revert. This would cause the escrow to pay out the incentives in `_handleAck()`, even though the message was never really delivered to the intended destination application.

Recommendation

- A small mitigation is to add code to `reemitAckMessage()` that checks that `implementationIdentifier` is a valid remote implementation (on the `sourceIdentifier` chain) of the escrow contract. While this may prevent re-emitting ack packets to invalid remote implementations, note that this will not prevent the example attack scenario described above. In any case, this mitigation will reduce the attack surface of the protocol.

```
function _handleAck(bytes32 destinationIdentifier, bytes memory
    destinationImplementationIdentifier, bytes calldata message, bytes32 feeRecipient
    , uint256 gasLimit) internal {
    bytes32 messageIdentifier = bytes32(message[MESSAGE_IDENTIFIER_START:
    MESSAGE_IDENTIFIER_END]);
    IncentiveDescription storage incentive = _bounty[messageIdentifier];
    // ...
    if (refundGasTo == address(0)) revert MessageAlreadyAcked();
    delete _bounty[messageIdentifier];  // The bounty cannot be accessed anymore.

    address fromApplication = address(bytes20(message[FROM_APPLICATION_START_EVM:
    FROM_APPLICATION_END]));
    bytes32 expectedDestinationImplementationHash = implementationAddressHash[
    fromApplication][destinationIdentifier];
    if (expectedDestinationImplementationHash != keccak256(
    destinationImplementationIdentifier)) revert InvalidImplementationAddress();

    bytes memory payload = abi.encodeWithSignature("receiveAck(bytes32,bytes32,bytes)
    ", destinationIdentifier, messageIdentifier, message[CTX1_MESSAGE_START: ]);
    bool success;
    assembly ("memory-safe") {
        success := call(maxGasAck, fromApplication, 0, add(payload, 0x20), mload(
    payload), 0, 0)
    }
```

Snippet 4.8: Relevant code in `_handleAck()` that executes the `receiveAck()` callback.

- Fix V-GNI2-VUL-002 so that the message identifier can be used to validate the destination chain identifier and destination chain escrow address.

Developer Response The developers changed `_messageDelivered` to be a nested mapping, where it must be indexed by the source identifier, the source escrow's address, and the message identifier. This ensures that a successful call to `reemitAckMessage()` requires the caller to provide the original source identifier, source escrow address, and ack (response) packet.

4.1.5 V-GNI2-VUL-005: Deadline of 0 is not rejected by _verifyTimeout()

Severity	Low	**Commit**	4abd57c
Type	Data Validation	**Status**	Fixed
File(s)	src/IncentivizedMessageEscrow.sol		
Location(s)	_verifyTimeout()		
Confirmed Fix At	https://github.com/catalystdao/GeneralisedIncentives/pull/40		

When a timeout packet is received by an escrow, the packet is validated by `_verifyTimeout()` before the timeout is executed and the incentives are paid out. The timeout packet contains a `deadline` field which is either zero (indicating that the message cannot be timed out) or a nonzero value indicating the Unix timestamp (in seconds) when the message is expired. Within `_verifyTimeout()`, there is a check that the `deadline` has elapsed to ensure that the message has legitimately expired. This check does *not* reject a `deadline` of zero.

```
// Do we need to check deadline again?
// Considering the cost: cheap, we will do it. In most instances it is not needed.
// This is because we must expect the remote implementation to also do the check to save gas
// since it is an obvious and valid remote check
uint64 deadline = uint64(bytes8(message[CTX2_DEADLINE_START:CTX2_DEADLINE_END]));
if (deadline >= block.timestamp) revert DeadlineNotPassed(deadline, uint64(block.timestamp));
```

Snippet 4.9: The deadline verification in `_verifyTimeout()`

Impact Assuming that only the current `IncentivizedMessageEscrow` implementation will be used for escrows, this has no security impact. A timeout packet may only be sent by `handleTimeout()` on the destination chain, which reverts if `deadline` is zero.

However, it may be possible for a future implementation of an escrow to send a timeout packet with a deadline of zero (e.g., perhaps by an implementation mistake). This would allow any message with a deadline of zero to be timed out immediately, even if the message has not been received yet on the destination. Full incentives will be paid out to the malicious relayer abusing this bug, and any legitimate ack packet for that message will be rejected by the source chain in the future.

Recommendation Add a `deadline == 0` check to `_verifyTimeout()` to ensure that the attack described above never occurs.

```solidity
// Load the deadline from the message.
uint64 deadline = uint64(bytes8(message[CTX0_DEADLINE_START:CTX0_DEADLINE_END]));

// Check that the deadline has passed AND that there is no opt out.
// This isn't a strong check but if a relayer is honest, then it can be used as a sanity check.
// This protects against emitting rouge messages of timeout before the message has had a time to execute IF deadline belong to messageIdentifier.
if (deadline == 0 || deadline >= block.timestamp) revert DeadlineNotPassed(deadline, uint64(block.timestamp));
```

Snippet 4.10: The deadline check in `timeoutMessage()`. Note that this will guard against a deadline of 0, unlike in `_verifyTimeout()`.

4.1.6 V-GNI2-VUL-006: Mock _proofValidPeriod inconsistent with documentation

Severity	Warning	**Commit**	4abd57c
Type	Maintainability	**Status**	Fixed
File(s)	`IncentivizedMockEscrow.sol`		
Location(s)	_proofValidPeriod()		
Confirmed Fix At	`https://github.com/catalystdao/GeneralisedIncentives/pull/40`		

The documentation of `_proofValidPeriod()` indicates that `block.timestamp` should be added to the duration where proofs remain valid in the implementation, so that the returned value is a timestamp. However, the mock implementation of this function in `IncentivizedMockEscrow` returns a constant corresponding to the time period without adding `block.timestamp` to the result.

```
1  * @notice Returns the duration for which a proof is valid for.
2  * It may vary by destination.
3  * @dev Remember to add block.timestamp to the duration where proofs remain vaild for
       .
4  * The setting needs to be sane: Do not set the proofValidPeriod to more than ~1
       years.
5  * @return timestamp The timestamp of when the application's message won't get
       delivered but rather acked back.
6  */
7 function _proofValidPeriod(bytes32 destinationIdentifier) virtual internal view
       returns(uint64 timestamp);
```

Snippet 4.11: Documentation comments on `IncentivizedMessageEscrow._proofValidPeriod()`

```
1     function _proofValidPeriod(bytes32 /* destinationIdentifier */) override internal
       view returns(uint64) {
2         return PROOF_PERIOD;
3     }
```

Snippet 4.12: Implementation of `IncentivizedMockEscrow._proofValidPeriod()`

Impact Future developers may reference the mock implementation as an example of how to implement an escrow contract. Since this implementation does not follow the rules described in the documentation and doesn't document this fact, developers may not realize that they need to add `block.timestamp` to the result of this function.

Recommendation Add `block.timestamp` to `PROOF_PERIOD` in this mock implementation, or document why it is missing from the implementation and that it is not safe to directly use this implementation from the mock contract.

Developer Response The developers note that this is actually a mistake in the documentation of `_proofValidPeriod()` (block.timestamp will actually be added to the result of `_proofValidPeriod()`when performing checks) and have updated the documentation accordingly.

4.1.7 V-GNI2-VUL-007: timeoutMessage() does not check validity of source chain

Severity	Warning	**Commit**	4abd57c
Type	Data Validation	**Status**	Intended Behavior
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	timeoutMessage()		
Confirmed Fix At	N/A		

The `timeoutMessage()` method may be invoked to force a timeout packet to be sent to a source chain. However, the method does not validate that the "to application" in the provided source-to-destination packet `message` corresponds to a valid remote implementation for the given `sourceIdentifier`.

```
function timeoutMessage(
    bytes32 sourceIdentifier,
    bytes calldata implementationIdentifier,
    uint256 originBlockNumber,
    bytes calldata message
) external payable virtual {
    bytes32 messageIdentifier = bytes32(message[MESSAGE_IDENTIFIER_START:
    MESSAGE_IDENTIFIER_END]);
    bytes32 storedAckHash = _messageDelivered[messageIdentifier];
    if (storedAckHash != bytes32(0)) revert MessageAlreadyProcessed();

    uint64 deadline = uint64(bytes8(message[CTX0_DEADLINE_START:CTX0_DEADLINE_END]));
    if (deadline == 0 || deadline >= block.timestamp) revert DeadlineNotPassed(
    deadline, uint64(block.timestamp));

    bytes memory receiveAckWithContext = bytes.concat(/* .. */);

    emit TimeoutInitiated(messageIdentifier);
    uint128 cost = _sendPacket(
        sourceIdentifier,
        implementationIdentifier,
        receiveAckWithContext,
        0
    );
```

Snippet 4.13: Lines of code that construct and send the packet. Note that no validation is performed on `sourceIdentifier` and `implementationIdentifier`.

In contrast, when `_handleMessage()` is called to process an incoming source-to-destination packet, it will perform the following validation:

```
address toApplication = address(bytes20(message[CTX0_TO_APPLICATION_START_EVM:
    CTX0_TO_APPLICATION_END]));
bytes calldata fromApplication = message[FROM_APPLICATION_LENGTH_POS:
    FROM_APPLICATION_END];

bytes32 expectedSourceImplementationHash = implementationAddressHash[toApplication][
    sourceIdentifier];
if (expectedSourceImplementationHash != keccak256(sourceImplementationIdentifier)) {
  // ... respond with error ...
  return /* ... */;
```



```
}
// ... send ack packet ...
```

Impact It may be possible to spoof the contextual destination chain details attached to a timeout packet. This may result in bugs and/or introduce attack vectors, depending on how much verification is performed in the concrete escrow implementation.

For example, suppose there are three escrow contracts A, B, C that are on separate chains. The two pairs (A, B) and (A, C) mark each other as remote implementations. Now, suppose that A sends a message to B, but the message is never delivered and times out. It is possible for some account to call `timeoutPacket()` on C with the message, causing a timeout packet to be sent from C to A. When A receives it, the extracted destination chain information will correspond to C's rather than B's. If the escrow implementation on A does not correctly validate the destination details, then the spoofed timeout packet may be accepted.

In the current implementation of `IncentivizedMessageEscrow`, there should be limited impact from spoofing the destination chain details. The `_verifyTimeout()` method (called when receiving a timeout packet) will check that the destination chain identifier corresponds to the message identifier, which will prevent such spoofed messages from being accepted.

Recommendation Although the impact of spoofing is limited, the attack surface can be reduced by adding code to `timeoutMessage()` that checks that `implementationIdentifier` is a valid remote implementation (on the `sourceIdentifier` chain) of the `toApplication` field in `message`. This code will likely be similar to the snippet in `_handleMessage()`.

Developer Response The developers note that the omission of the check is intended behavior:

> We want to be able to timeout all messages, including messages sent to the wrong implementation. If we implemented this check, then we wouldn't be able to process messages were the proof is lost and were un-authorised.

4.1.8 V-GNI2-VUL-008: Code Duplication between _handleAck and _handleTimeout

Severity	Warning	**Commit**	4abd57c
Type	Maintainability	**Status**	Acknowledged
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_handleTimeout, _handleAck		
Confirmed Fix At	N/A		

There is significant shared complex logic between the implementation of `IncentivizedMessageEscrow._handleAck()` and `IncentivizedMessageEscrow._handleTimeout()`.

```
bytes memory payload = abi.encodeWithSignature("receiveAck(bytes32,bytes32,bytes)",
    destinationIdentifier, messageIdentifier, message[CTX1_MESSAGE_START: ]);
bool success;
assembly ("memory-safe") {
    success := call(maxGasAck, fromApplication, 0, add(payload, 0x20), mload(payload)
    , 0, 0)
}
unchecked {
    if (!success) if(gasleft() < maxGasAck * 1 / 63) revert NotEnoughGasExeuction();
}
```

Snippet 4.14: Example of the shared logic in `_handleAck()`. The only difference with the similar code in `_handleTimeout()` is that the latter will prepend an error code to the message argument.

Impact If the developers want to change how the low level calls to the application are performed, the restrictiveness of the gas remaining check, etc, they will have to update the implementation in multiple places, increasing the risk of forgetting to update one of the locations.

Recommendation Factor out the shared logic into a helper function to keep all the documentation and the implementation in a single place.

Developer Response The developers acknowledge the issue but not plan to resolve it because "it is not possible to implement gas efficiently."

4.1.9 V-GNI2-VUL-009: Unvalidated assumption that timeoutMessage() packet is SOURCE_TO_DESTINATION

Severity	Warning	**Commit**	4abd57c
Type	Data Validation	**Status**	Fixed
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	timeoutMessage()		
Confirmed Fix At	`https://github.com/catalystdao/GeneralisedIncentives/pull/40`		

When a destination chain account invokes `timeoutMessage()` to queue a timeout packet to be delivered to the source chain, the account must provide the original source-to-destination message that has timed out as the `message` function parameter. However, the `timeoutMessage()` method assumes that the provided packet is a source-to-destination packet without validating that it indeed is one.

```
// Load the deadline from the message.
uint64 deadline = uint64(bytes8(message[CTX0_DEADLINE_START:CTX0_DEADLINE_END]));

// Check that the deadline has passed AND that there is no opt out.
// This isn't a strong check but if a relayer is honest, then it can be used as a sanity check.
// This protects against emitting rouge messages of timeout before the message has had a time to execute IF deadline belong to messageIdentifier.
if (deadline == 0 || deadline >= block.timestamp) revert DeadlineNotPassed(deadline, uint64(block.timestamp));
```

Snippet 4.15: Lines of code that assume that `message` is a source-to-destination packet.

Impact It is possible for a destination chain account to accidentally provide an ack packet or timeout packet as the `message` parameter. Since the position of the `deadline` field in the source-to-destination packet would be contained in the message field of the ack and timeout packets, such a mistake can likely cause the deadline check to be bypassed (e.g., arbitrary bytes causing the deadline to be "large") and therefore cause a timeout packet to be sent to the specified source chain. This flaw may also be a potential attack vector for exploiting vulnerabilities in the timeout packet verification logic in `processPacket()` (such as the example described in V-GNI2-VUL-007), although the auditors have not identified any such concrete exploits at the time of writing.

```
bytes memory receiveAckWithContext = bytes.concat(
    CTX_TIMEDOUT_ON_DESTINATION,
    messageIdentifier,
    message[FROM_APPLICATION_LENGTH_POS:FROM_APPLICATION_END],
    bytes8(deadline),
    bytes32(originBlockNumber),
    message[CTX0_MESSAGE_START: ]
);
```

Snippet 4.16: Lines of code that construct the timeout packet

Recommendation As an additional "sanity check", insert a require statement checking that `message[CONTEXT_POS] == CTX_SOURCE_TO_DESTINATION`.

Developer Response The developers applied the recommendation.

4.1.10 V-GNI2-VUL-010: To-application address length not validated

Severity	Warning	**Commit**	4abd57c
Type	Data Validation	**Status**	Acknowledged
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_handleMessage()		
Confirmed Fix At	N/A		

When `_handleMessage()` is called to process an incoming source-to-destination packet, the `toApplication` is read from `message[CTX0_TO_APPLICATION_START_EVM:CTX0_TO_APPLICATION_END]`. However, no validation is performed to check that the address length matches the length of an EVM address.

```
// Deliver message to application.
// Decode gas limit, application address and sending application.
address toApplication = address(bytes20(message[CTX0_TO_APPLICATION_START_EVM:
    CTX0_TO_APPLICATION_END]));
bytes calldata fromApplication = message[FROM_APPLICATION_LENGTH_POS:
    FROM_APPLICATION_END];

// Check if the message is valid. This includes:
// - Checking if the sender is valid.
// - Checking if the message has expired.

bytes memory acknowledgement;

bytes32 expectedSourceImplementationHash = implementationAddressHash[toApplication][
    sourceIdentifier];
// Check that the application allows the source implementation.
// This is not the case when another implementation calls this contract from the
    source chain.
// Since this could be a mistake, send back an ack with the relevant information.
if (expectedSourceImplementationHash != keccak256(sourceImplementationIdentifier)) {
```

Snippet 4.17: Snippet in `_handleMessage()` that reads and validates `toApplication`.

Impact This currently has no security impact. If a legitimate source-to-destination packet meant for an non-EVM chain is mistakenly sent to an `IncentivizedMessageEscrow` on an EVM-chain, the escrow will silently truncate the application address before running the validation code. Thus, the packet will be considered non-authenticated (and therefore rejected as it should be), assuming that the escrow contract does not have an application whose address is equal to the truncated address specified in the packet (equality being an unlikely scenario).

Recommendation Add a check that `message[CTX0_TO_APPLICATION_LENGTH_POS] == 20` (an Ethereum address is 20-bytes long) and handle the error appropriately. For example, because the target is meant for the wrong chain, the authentication check could include the length check as an additional criterion.

Developer Response The developers acknowledge the issue but do not intend to fix it as of the time of writing:

> While introducing a dedicated check and error code for this could be done, assuming everything is setup correct, there aren't any side effects from the current implementation.

We prefer the simplified implementation of no check.

4.1.11 V-GNI2-VUL-011: Ack packet sent instead of timeout packet when received message is past deadline

Severity	Warning	Commit	4abd57c
Type	Logic Error	Status	Intended Behavior
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_handleMessage()		
Confirmed Fix At	N/A		

When a message is received on the destination chain `IncentivizedMessageEscrow` after the message's timeout has elapsed, an ack packet is sent back to the source chain escrow indicating that the message has timed out. This is handled differently from the case where a destination chain account will call the `timeoutMessage()` method, causing a timeout packet to be sent back.

```
// Check that if the deadline has been set (deadline != 0). If the deadline has been set,
// check if the current timestamp is beyond the deadline and return TIMED_OUT if it is.
uint64 deadline = uint64(bytes8(message[CTX0_DEADLINE_START:CTX0_DEADLINE_END]));
if (deadline != 0 && deadline < block.timestamp) {
    acknowledgement = bytes.concat(
        MESSAGE_TIMED_OUT,
        message[CTX0_MESSAGE_START: ]
    );

    // Encode a new message to send back. This lets the relayer claim their payment.
    // Incase of timeouts, we give all of the gas.
    receiveAckWithContext = bytes.concat(
        bytes1(CTX_DESTINATION_TO_SOURCE),  // Context
        messageIdentifier,  // message identifier
        fromApplication,
        feeRecipient,
        bytes6(uint48(2**47)),  // We set the gas spent as max. Note that 2**47 is likely to be larger than maxGas but it is fine since we are doing max on source.
        bytes8(uint64(block.timestamp)),  // If this overflows, it is fine. It is used in conjunction with a delta.
        acknowledgement
    );

    // Store a hash of the acknowledgement so we can later retry a potentially invalid ack proof.
    _messageDelivered[messageIdentifier] = keccak256(receiveAckWithContext);

    // Message has been delivered and shouldn't be executed again.
    emit MessageDelivered(messageIdentifier);
    return receiveAckWithContext;
}
```

Snippet 4.18: Snippet from `_handleMessage()` that handles an expired message

The incentives for timeout packets are paid out differently than that for ack packets. Specifically, the escrow pays out incentives to both relayers when receiving a valid ack packet. For a timeout packet, the escrow will only pay out the relayer of the timeout packet.

Impact It is not clear whether the current behavior in `_handleMessage()` is intended. The source-to-destination chain relayer is rewarded as though the message didn't time out, even though they delivered the message too late (and therefore did not execute the desired transaction). Arguably, this is unfair to the application, which had created the incentives so that the message *would* be delivered on time and executed. Due to the reward being maintained, there is less incentive for source-to-destination relayers to ensure that the message arrives on time.

Recommendation Treat messages that are delivered after the deadline has elapsed the same way as messages that are timed out using `timeoutMessage()`.

Developer Response The developers note that the current behavior is intentional:

> If we required a timeout packet to be sent (rather than an ordinary package with a special fail flag), then it would complicate a relayer implementation. Notice that the TimeoutMessage function contains a lot of information which cannot be found within the relayed package. This leaves us with 2 options:
>
> 1. Always require TimeoutMessage to be called from a relayer (and say the associated complexity is just how it is)
> 2. Define a way for relayer to easily timeout a message as if nothing happened.
>
> We choose option 2 because it simplifies A LOT of relaying assumptions like: What happens if you relay right on the edge of the deadline? You get paid normally if it executes before deadline and more if it executes after Do you risk a failing transaction? Nope Getting paid less than expected? You "risk" being paid more.

They further note that the timeout packet needs to be handled specially compared to the ack packet:

> TimeoutMessage is intended to be manually invoked. Why? Because we cannot verify that any timeoutMessage is valid. Rather, we can first validate that a timeoutMessage was valid once it is verified on the destination. As a result, we cannot block any other timeout from being emitted, so someone could emit a timeout with another specified relayer. We have thought deep and hard about a way to make TimeoutMessage blocking (only 1 timeout can be emitted) but it is not possible since it is emitted optimistically.

4.1.12 V-GNI2-VUL-012: Timeout packet awards delivery incentives to the ack relayer

Severity	Warning	**Commit**	4abd57c
Type	Logic Error	**Status**	Intended Behavior
File(s)	`IncentivizedMessageEscrow.sol`		
Location(s)	_handleTimeout()		
Confirmed Fix At	N/A		

When the source chain escrow processes a timeout packet in `_handleTimeout()`, the relayer of the timeout packet is awarded the full incentive—including the amount allocated for delivering the original message to the destination—even though that message was never delivered.

```
(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
);
```

Snippet 4.19: The call to `_payoutIncentive()` in `_handleTimeout()` that awards the incentives and refunds gas fees.

Impact This will minimize the application's refund because the gas allocated for the delivery of the message to the source chain was not used, but it is still paid out to the timeout packet relayer. One might argue that this is unfair to the application, because the application's intended goal (successful delivery of the message) was not achieved.

Recommendation Since the message was never delivered, refund the incentives allocated for the source-to-destination relayer back to the application.

Developer Response The developers note that the current behavior is intentional:

> We want to reward the person who went out of their way (remember, there is kind of an assumption that the original proof was lost or something else broke down) to emit a timeout. This also have to be standardized with the timeout implementation.

With respect to the incentive amount, the developers note:

> This has actually been something I thought a lot amount. Do we pay out the full amount, do we pay out less, etc.

If a deadline is set, it is likely that there is some kind of urgency involved. If the message didn't arrive in time it was likely because the incentive was too small. By paying out a larger incentive on timeout, it is more likely that that timeout will be sent back. Yes, there are counterpoints like: Not all AMBs support unbounded deadlines.

4.1.13 V-GNI2-VUL-013: Typos and Comment Clarifications

Severity	Info	**Commit**	4abd57c
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See issue description		
Confirmed Fix At	https://github.com/catalystdao/GeneralisedIncentives/pull/34		

Description In the following locations, the auditors identified minor typos and potentially misleading comments:

- `src/interfaces/IMessageEscrowEvents.sol`:
  - Line 13: This comment is missing the reason why this event isn't indexed

```
event TimeoutInitiated(bytes32 messageIdentifier); // Not indexed since
    relayers can
```

- `README.md`
  - Line 115: verifiable and message are misspelled

```
Not all AMBs support perpetually verifable proofs. To avoid loss of
    packages, timeouts are implemented. When submitting messages,
    applications can specify a timestamp where if a message is delivered
    after that, the destination application will not receive it. Instead an
     ack with failure code ‘0xfe‘ will be send back. Additionally, if no
    mesasge has been delivered a timeout attempt can be sent back.
```

  - Line 121: messageTimedout is misspelled

```
The messages structure can be found in src/MessagePayload.sol. 3 message
    types are defined: ‘SourcetoDestination‘, ‘DestinationtoSource‘, and ‘
    essageTimedout‘ which can be identified by the first byte of the
    message as 0x00, 0x01, or 0x02 respectively.
```

- `src/IncentivizedMessageEscrow.sol`
  - Lines 105-106: This sentence is confusing because it contains three negations. It should be reworded to something like: "It should only be set to a contract which implements either a fallback or a receive function that never reverts."

```
    /**
     * @param sendLostGasTo Who should receive Ether which would otherwise
    block
     * execution? It should never be set to a contract which does not
    implement
     * either a fallback or receive function which never revert.
     * It can be set to address 0 or a similar burn address if no-one wants
     to take ownership of the ether.
     */
    constructor(address sendLostGasTo) {
```

  - Line 237: There is a space missing in the words "times out"

```
     * Furthermore, if the package timesout there is no gas refund.
```

- Line 257: Missing the description of which thing to needs to be checked. Should be something like: "checking if the length of the destinationImplementation entry for that application is not 0"

```
        // Check that the application has set a destination implementation
    by checking if the is not 0.
```

- Line 404: This comment should be updated because this section of code no longer decodes the gas limit

```
        // Deliver message to application.
        // Decode gas limit, application address and sending application.
        address toApplication = address(bytes20(message[
    CTX0_TO_APPLICATION_START_EVM:CTX0_TO_APPLICATION_END]));
        bytes calldata fromApplication = message[
    FROM_APPLICATION_LENGTH_POS:FROM_APPLICATION_END];
```

- Line 545: This sentence is incomplete

```
        // Ensure the bounty can only be claimed once. This call is matched
     on the ack side,
        // so it also ensures that an ack cannot be delivered if a timeout
    has been
        if (refundGasTo == address(0)) revert MessageAlreadyAcked();
```

- Line 705: Badly should be replaced with bad

```
// For timeouts, this could fail because of fradulent sender or badly data.
```

- Line 715: entirety is misspelled

```
// The entirty of the incoming message is untrusted. So far we havn't done
    any verification of
```

- Line 798: The comment indicates that the logic should end at this if statement for timeouts because the `targetDelta` for a timeout packet is set to 0. However, the call to `_payoutIncentive()` in `_handleTimeout()` sets the `destinationFeeRecipient` and the `sourceFeeRecipient` to the same address, so the logic will always end at the above if statement.

```
        // If both the destination relayer and source relayer are the same
    then we don't have to figure out which fraction goes to who.
        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. For
    timeouts, logic should end here.
        if (targetDelta == 0) {
```

- Line 1007: Language is misspelled

```
// To maintain a common implementation langauge, emit our event before
    message.
```

Impact These minor errors may lead to future developer confusion.

Glossary

AMB Arbitrary Message Bridge. 1