MiloTruck

Shape

Security Review

September 7, 2026

Contents

1 Introduction

1.1 About MiloTruck

MiloTruck is an independent security researcher, primarily working as a Lead Security Researcher at Spearbit and Cantina. Previously, he was part of the team at Renascence Labs and a Lead Auditor at Trust Security.

For private audits or security consulting, please reach out to him on Twitter *@milotruck*.

1.2 Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

2 Risk Classification

Severity Level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality.
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality/availability.
- Low - Funds are **not** at risk.

2.2 Likelihood

- High - Highly likely to occur.
- Medium - Might occur under specific conditions.
- Low - Unlikely to occur.

3 Executive Summary

3.1 About Gasback

Shape is an Ethereum L2 for NFTs, purpose-built with decentralization at its core.

For more information, visit `https://shape.network/`.

3.2 Overview

Project Name	Shape
Project Type	Bridge
Language	Solidity
Repository	Vectorized/gasback
Commit Hash	07b7a27808499bd81fd09fd8d9de54caf16ad463

3.3 Issues Found

High	0
Medium	1
Low	2
Informational	3

4 Findings

4.1 Medium Risk

4.1.1 L1 to L2 deposit transactions could refund more ETH than gas burned

Context: Gasback.sol#L154-L155

Description: The `Gasback` contract burns gas and refunds ETH to the caller:

```
uint256 ethFromGas = gasToBurn * block.basefee;
uint256 ethToGive = (ethFromGas * $.gasbackRatioNumerator) / GASBACK_RATIO_DENOMINATOR;
```

An important invariant is that callers should never earn more ETH than the amount spent burning gas. However, this invariant could be broken for L1 -> L2 deposit transactions.

According to the Optimism specs, for L1 -> L2 deposit transactions, gas is bought on L1 and has its own price based on the fee market in `OptimismPortal`. Therefore, it is entirely possible for the price paid for gas on L1 to be lower than `block.basefee` on L2, which would make `ethFromGas` higher than what the caller actually paid.

Recommendation: A possible solution would be to revert if `tx.gasprice == 0` since only L1 -> L2 deposit transactions can have a zero gas price, as seen in the latest L1 -> L2 transaction on Optimism.

Shape: Fixed in PR 20 as recommended.

MiloTruck: Verified.

4.2 Low Risk

4.2.1 Push-payment pattern in `FeeVaultSplitter` allows malicious payees to grief the `Gasback` contract

Context: FeeVaultSplitter.sol#L43-L50

Description: When `FeeVaultSplitter` receives ETH, it automatically sends the ETH received to payees. However, from a security perspective, this pattern introduces a lot of ways for a malicious payee to grief the `Gasback` contract (apart from the OOG DOS already mentioned in the comments).

For example, if a malicious payee is ordered after the `Gasback` contract in `payees_`, he can re-enter the `Gasback` contract to use up the ETH distributed to it first. Therefore, the caller who called the `Gasback` contract originally will not have any ETH leftover, and his call will be skipped.

There are also other ways to make a call to `Gasback`'s fallback function pass through without reverting.

Also, even if payees are not malicious, a payee with some functionality in their receive function is *technically* stealing gas from whoever calls the `Gasback` contract. This violates the following line in the RIP-7767 specification (but I'm kind of nitpicking here):

> The actual amount of gas burned may vary but SHOULD be proportional to `gasToBurn`.

Recommendation: A possible solution would be to remove the `_distribute()` functionality and make the `Gasback` contract call `release()` for itself directly, but this is quite a huge refactor.

Shape: Acknowledged. We consider the payees trusted as they are all associated with operating the chain itself, and would likely be able to create much worse harm to the chain than calling the release and reentering.

MiloTruck: Acknowledged.

4.2.2 Current `FeeVault` configuration is incompatible with `Gasback`

Context: Gasback.sol#L113-L118

Description: `triggerBaseFeeVaultWithdraw()` calls `FeeVault.withdraw()` as seen below:

```
    function triggerBaseFeeVaultWithdraw(uint256 expectedSelfBalanceAfter) external onlySelf {
        (bool success,) =
            _getGasbackStorage().baseFeeVault.call(abi.encodeWithSignature("withdraw()"));
        require(success);
        require(address(this).balance >= expectedSelfBalanceAfter);
    }
```

`FeeVault.withdraw()` has a minimum withdrawal amount check:

```
require(
    address(this).balance >= MIN_WITHDRAWAL_AMOUNT,
    "FeeVault: withdrawal amount must be greater than minimum withdrawal amount"
);
```

`MIN_WITHDRAWAL_AMOUNT` is currently set to `0.01e18`, which should be a lot higher than the average `ethToGive`.

Assume `Gasback.balance < ethToGive < BaseFeeVault.balance < MIN_WITHDRAWAL_AMOUNT`. This means the `Gasback` contract has insufficient ETH and needs to pull from `BaseFeeVault`, which has enough ETH to cover `ethToGive` but less than `MIN_WITHDRAWAL_AMOUNT`. `withdraw()` would revert although there is enough ETH to pay the caller.

Additionally, `WITHDRAWAL_NETWORK` is currently set to `WithdrawalNetwork.L1`, but it needs to be `WithdrawalNetwork.L2` to send funds to `Gasback`.

Recommendation: Lower `MIN_WITHDRAWAL_AMOUNT` and switch `WITHDRAWAL_NETWORK` to `WithdrawalNetwork.L2`.

Shape: Acknowledged, we will lower this during mainnet deployment. We also plan on switching `WITHDRAWAL_NETWORK` to L2 during deployment.

MiloTruck: Acknowledged.

4.3 Informational

4.3.1 `release()` in `FeeVaultSplitter` isn't protected by `nonReentrant`

Context: FeeVaultSplitter.sol

Description: Functions in `FeeVaultSplitter` have `nonReentrant` to protect against reentrancy.

However, `release()` isn't protected by `nonReentrant`, so the reentrancy guard doesn't completely prevent reentrancy. And it cannot be overridden with a `nonReentrant` modifier since `release()` is called in `_distribute()`.

However, this does not seem to be exploitable.

Recommendation: A partial solution would be to override the `release()` function for tokens to always revert to reduce the attack surface.

Shape: Acknowledged. The `release()` function for tokens is left as is so we have the option to handle ERC20 in the future.

MiloTruck: Acknowledged.

4.3.2 Success is not checked for `triggerBaseFeeVaultWithdraw()`

Context: Gasback.sol#L161-L165

Description/Recommendation:

The fallback function of the `Gasback` contract does not check if `triggerBaseFeeVaultWithdraw()` succeeded when calling it:

```
  assembly {
      mstore(0x00, 0xc70746b1) // `triggerBaseFeeVaultWithdraw(uint256)`.
      mstore(0x20, ethToGive)
      pop(call(gas(), address(), 0, 0x1c, 0x24, 0x00, 0x00))
  }
```

However, this is safe as the contract would have insufficient ETH anyways if the call failed.

Shape: Fixed in PR 20 by adding a comment about this.

MiloTruck: Verified.

4.3.3 Configuration values in the constructor will not be set with an EIP-7702 delegated EOA

Context: Gasback.sol#L49-L54

Description: The constructor of the `Gasback` contract is as such:

```
constructor() payable {
    GasbackStorage storage $ = _getGasbackStorage();
    $.gasbackRatioNumerator = 0.6 ether;
    $.gasbackMaxBaseFee = type(uint256).max;
    $.baseFeeVault = 0x4200000000000000000000000000000000000019;
}
```

These configuration values will not be set if the contract is used with an EIP-7702 delegated EOA. Their setters must be called individually during deployment, as seen in `Delegate7702.s.sol`.

Also, note that there is a risk of an attacker front-running the deployer and calling `Gasback` before the setters are called (since Foundry scripts broadcasts multiple transactions). However, it does not seem possible to exploit this as:

- If `gasbackRatioNumerator = 0`, `ethToGive = 0`.
- If `gasbackMaxBaseFee = 0`, the fallback function will perform a pass through.
- If `baseFeeVault = address(0)` when it is called, it means the contract has insufficient ETH and a pass through will occur.

Recommendation: Consider leaving a comment to call the setters here, for future integrators.

Shape: Fixed by adding the comment in PR 20.

MiloTruck: Verified.