certora

Security Assessment & Formal Verification Final Report

Atlas

December 2024

Prepared for

FastLane Labs

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Atlas	https://github.com/FastLane-Labs/atlas	[8127c0152aac7161d8aa1e3b69d8adc24e2755b4](8127c0152aac7161d8aa1e3b69d8adc24e2755b4)	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of **Atlas** using the Certora Prover and manual code review findings. The work was undertaken from **November 19, 2024** to **December 20, 2024**.

The following contract list is included in our scope:

```
src/contracts/atlas/*
src/contracts/common/*
src/contracts/dapp/*
src/contracts/gasCalculator/*
src/contracts/helpers/Simulator.sol
src/contracts/helpers/Sorter.sol
src/contracts/libraries/*
src/contracts/types/*
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Please note that a few more formal rules are not included in this report, as they were proven with an unreleased version of the Certora Prover. Once those rules are proven on a released version of the Certora Prover, we will add them to the next version of this document.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0		
High	6	0	6
Medium	3	0	3
Low	4	1	3
Informational	6	2	4
Total	**19**	**3**	**16**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
H-01	Repeatable Temporary Freeze of User Assets	High	Fixed
H-02	Escrow duration can be bypassed and bonded balance can be withdrawn immediately	High	Fixed
H-03	_bundlerSurcharge will always be less than _maxBundlerRefund	High	Fixed
H-04	Bundler can get unfairly penalized and loses funds to winning solver due to gas limit	High	Fixed
H-05	Solver can frontrun bundler and make sure he becomes the winning solver by DOSing other solvers and allow MEV-like behavior	High	Fixed
H-06	Bundler-solvers can exploit users by setting unrealistic high gas limit	High	Fixed
M-01	Bundler wrongly pays Atlas surcharge when operation fails in certain situations	Medium	Fixed

M-02	Atlas might not receive storage refund	Medium	Fixed
M-03	Solver might fail when it should succeed due to incorrect bonded amount	Medium	Fixed
L-01	Remove Bundler surcharge in _updateAnalytics	Low	Fixed
L-02	Governance signatory behaves unexpectedly	Low	Fixed
L-03	False Update to Analytics	Low	Fixed
L-04	PerBlockLimit can lead to accounting exploit on Bundlers or Solvers	Low	Acknowledged

High Severity Issues

H-01 Repeatable Temporary Freeze of User Assets		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

Attacker can cause a solver to not be able to withdraw his funds. By crediting the solver 1 wei with malicious bundler and update the solver's lastAccessedBlock

and the solver's timelock resets. Whenever the timelock is going to end, the attacker will give solver 1 wei with the _credit function so the solver cant ever withdraw his funds.

Recommendation:

Like in HIGH [5], remove _aData.lastAccessedBlock = uint32(block.number); from the _credit function. If the team wants to keep the functionality of coupling the solvers' winning actions, add it in a different place.

Customer's Response:

Fixed in commit [fae5ba5]

H-02 Escrow duration can be bypassed and bonded balance can be withdrawn immediately		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

The unbonding timelock exists to offer stability to off-chain calculations. However, it is possible to bypass this timelock, leading to off-chain calculations to fail and bundles to behave unexpectedly potentially causing a loss of funds to the Bundler because it would not get the gas refund as expected.

This is possible because a malicious actor could withdraw the atlETH without timelock by running their own protocol and bundler. By calling the borrow function as the winning solver, they can send the ETH away, and repay with the bonded amount, thus withdrawing without the unbonding timelock (and a very small fee). Usually they would need to wait or pay the atlas and bundler fees, but now they don't need to do either.

Recommendation:

Borrowed funds should be paid back on top of the bonded balance. More explicitly, the bonded balance should be considered only for the execution of the operation, not as a way to repay borrowed funds. Instead borrowed funds should be repaid through a different accounting system. This would prevent malicious actors from bypassing the unbonding period.

Customer's Response:

Fixed in commit [4d5c97b]

H-03 _bundlerSurcharge will always be less than _maxBundlerRefund		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

In GasAccounting.sol if (_bundlerSurcharge > _maxBundlerRefund) does not make sense during the execution flow that goes through _adjustAccountingForFees() function when there are no winning solvers. This is because _bundlerSurcharge will always be less than _maxBundlerRefund. Max bundler refund is 80% of the metacall gas cost and the _bundlerSurcharge is much less than 80% so this branch will never happen and bundlers will get this extra refund when atlas should get it.

Recommendation:

We recommend removing the if statement (_bundlerSurcharge > _maxBundlerRefund) { as it can never be reached (unless the bundler surcharge is unusually high). We propose to simply reward 20% of the original gas cost to Atlas and give the remaining to the bundler.

Customer's Response:

Fixed during the refactor of the Gas Accounting system, in commit [fba1c9e]

H-04 Bundler can get unfairly penalized and loses funds to winning solver due to gas limit		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

Taking in consideration the _adjustAccountingForFees() function (lines 387-404), let's assume we have 10 solvers. Each of them requests solverGasLimit as gas limit (and they have the bond to pay) but when making the actual call 9 of them revert and don't spend any gas. The last solver is successful. Now gasLeft is at least 9 * solverGasLimit but the _upperGasRemainingEstimate is 1 * solverGasLimit. The bundler will be penalized for this but it is not their fault.

Recommendation:

We recommend either dynamically deriving solverGasLimit from the total bonded among solvers or subtracting gasLeft from the gas limit of solvers that have reverted (have not spent any gas) or solvers that spent a significantly smaller percentage of it.

Customer's Response:

Fixed in commit [8de7b0d]

H-05 Solver can frontrun bundler and make sure he becomes the winning solver by DOSing other solvers and allow MEV-like behavior		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

In the credit function there is an update to _aData.lastAccessedBlock = uint32(block.number);.

A malicious solver can frontrun the bundler call and create a MEV-like arbitrage (with the userOp the arbitrage is big). Usually the solvers will bid the highest amount and extract the value the MEV attacker placed to create the arbitrage. But a malicious solver can frontrun the metacall and give small amounts (even 0) of gas to each solver in the bundle and make sure their next solverOp in the same block will revert.

For example by permissionlessly creating a scenario where all solvers failed the gasRefundBeneficiary gets credited.

The malicious solver can call Atlas with their own protocol and use the gasRefundBeneficiary address as the honest solver to DOS. Now in this block this solver cannot run. We can repeat this for all the solvers in the real bundle so the malicious solver is the only one left. Now the attacker can do this backrun with a very small bid (1 wei) and keep the rest of the profit as if Atlas does not exist.

This allows a **MEV-like** behavior which guarantees to the **MEV** that he will be the backrunner and thus allow him to commit the **sandwich attack**, this is exactly what Atlas should **prevent**.

Recommendation:

Remove _aData.lastAccessedBlock = uint32(block.number); from the _credit function. If you want to prevent solver from operating twice in that same block make sure that Atlas counts only operations that the solver approved.

Customer's Response:

Fixed in commit [fae5ba5]

H-06 Bundler-solvers can exploit users by setting unrealistic high gas limit		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

Malicious bundler-solvers can send a high gas limit and because the _bundlerGasOveragePenalty goes to the winning solver there is no real penalty when the bundler is the winning solver.

Consider a bundler-solver with a high bond and the worst bid (1 wei) that sends the rest of the profit to an external address.

The attacker can give a very high gas limit for the tx. All the solvers fail and the winning solver (the bundler) is the one with the high bond. They paid the smallest fee possible and didn't get punished. Thus stealing all profits from Atlas and the user.

Recommendation:

Allow DappControl to establish a gas limit operation rather than having a gas limit that can be arbitrarily set by the bundler.

Customer's Response:

Fixed in commit [9fbf40d]()

Medium Severity Issues

M-01 Bundler wrongly pays Atlas surcharge when operation fails in certain situations		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

Atlas surcharge is always included when t_writeoffs is increased.

Consequently, if the operation fails due to the bundler's fault or it fails due to solvers not having enough bonded amount, the bundler will end up paying for the Atlas surcharge. In the case of _bidFindingIteration the bundler is responsible for the onchain validation and there is a writeoff by _writeOffBidFindGasCost but Atlas still gets the surcharge on the bundler expense.

Recommendation:

In _adjustAccountingForFees() find atlasSurchargeWriteoffs from t_writeoffs then:

netAtlasGasSurcharge -= atlasSurchargeWriteoffs

adjustedWriteoffs -= atlasSurchargeWriteoffs

Customer's Response:

Fixed during the refactor of the Gas Accounting system, in commit [fba1c9e]

M-02 Atlas might not receive storage refund		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

If t_solverSurcharge is zero because all the solvers have failed and have bonded just exactly enough to cover their gas but nothing more, than the gas refund also goes to the bundler because all the logic that takes the refund from the bundler is inside the if block that happens when t_solverSurcharge is not zero.

Recommendation:

We believe the bonded amount should already consider the fees for the solver. Change Escrow.sol line 353 to:

if (_gasCost.withSurcharges(ATLAS_SURCHARGE_RATE, BUNDLER_SURCHARGE_RATE) < _solverBalance) {

Customer's Response:

Fixed during the refactor of the Gas Accounting system, in commit fba1c9e

M-03 Solver might fail when it should succeed due to incorrect bonded amount		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Escrow.sol	Status: Fixed	Violated Property: None

Description:

When assessing whether the solver has enough balance to pay for the transaction, the surcharges for Atlas and Bundler are not accounted for. This could lead to a Solver running out of balance, even though theoretically their bonded amount should have been enough.

Recommendation:

We recommend the same fix as MEDIUM [2]:

Change Escrow.sol line 353 to:

```
if (_gasCost.withSurcharges(ATLAS_SURCHARGE_RATE, BUNDLER_SURCHARGE_RATE) < _solverBalance) {
```

Customer's Response:

Fixed in commit 5f53a6a

Low Severity Issues

L-01 Remove Bundler surcharge in _updateAnalytics		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: GasAccounting.sol	Status: Fixed	Violated Property: None

Description:

When _updateAnalytics is called through _assign() we just need to remove the surcharge or change the name of that variable and add the Atlas surcharge.

Customer's Response:

Fixed in commit c302995

L-02 Governance signatory behaves unexpectedly

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: DappIntegration.sol DappControl.sol	Status: Fixed	Violated Property: None

Description: The DappControl.sol governance can call DappIntegration.removeSignatory() and remove themselves as a signatory. This would make it impossible to transfer governance via the DappControl.acceptGovernance() function, because as the execution function reaches DappIntegration.changeDAppGovernance() it would revert, as the signatory would not longer exist. This means the governance would still be in control, even though it should not, as it has been removed as a signatory. If it is desired to completely remove governance, we suggest making it so DappControl.governance also gets changed if it removes itself as a signatory.
This would prevent the case where governance falsely removes itself, in an attempt to high-jack the intended newGovernance from calling acceptGovernance(), which could happen if there is an untimely key compromise.

Customer's Response:

Fixed in commit 0e114a5

L-03 False Update to Analytics		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: GasAccounting.sol	Status: Fixed	Violated Property: None

Description:

The gasRefundBeneficiary is marked as a winner in the auctionWins when all the solvers failed. If a solver is also a gasRefundBeneficiary and all the solvers fail then he gets +1 in the auctionWins and in the auctionFails. There is double counting of the totalGasValueUsed in that case because when the fail happens we increase the totalGasValueUsed for the solver run and we count that again in the _credit inside _settle .

Customer's Response:

Fixed in commit [c302995]

certora

L-04 PerBlockLimit can lead to accounting exploit on Bundlers or Solvers		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: Escrow.sol	Status: Acknowledged	Violated Property: None

Description:

Scenario 1: Evil Auctioneer could make Solvers lose money

An Auctioneer knowingly replays solverOps and Solver has to pay for it due to _validateSolverOpDeadline() and bundlersFault() logic.

Malicious Auctioneer could replay solverOp in the same bundle and the solver would have to pay for it according to _handleSolverAccounting() logic (plus other factors mentioned above). We believe the bundler should be the one to blame if lastAccessedBlock >= block.number(Escrow line 387). Because they should know better than to replay the same solverOps. While the Solver has no control over it. Consequently Bundlers can cause Solvers to lose money either willingly or by mistake.

Scenario 2: Evil Solver could make Bundlers lose money

Solver submits two operations, the first uses all his bonded/unbonding balance, leaving the second one with no balance to subtract from. Bundler will have to pay for this deficit, which could be big if depending on solverOps cost.

Recommendation:

Blame the bundler if there are two solverOps from the same msg.sender in the same bundle.

This aligns incentives with Bundlers and makes it so scenario 2 will not happen in practice, and if this happens it can purely be attributed to Bundler incompetence.

We can further enforce on-chain by looping over the bundle and checking each solverOp to see if there is a repeating msg.sender in it.

Customer's Response:

Acknowledged. No changes made as we still feel blaming solvers for the PerBlockLimit error is the best balance given the system's incentives and sophistication of the auctioneer, bundler, and solver parties, as well as the existing trust assumptions around the auctioneer.

To address scenario 1: We assume solvers to be sophisticated actors in this system. If an auctioneer acts dishonestly such as attempting to replay solverOps, we expect solvers to quickly notice and stop participating in those metacalls. The downside risk to solvers is also fairly low as their solverOp would not execute and the gas used would be minimal. We cannot fix this by shifting blame to the bundler as other bundlers may be coincidentally including the same solver in a different metacall in the same block, which is something even a sophisticated bundler wouldn't be able to detect, so cannot be attributed to bundler incompetence.

To address scenario 2: The auctioneer has the ability to exclude the 2nd solver from the metacall when their simulation of the solverOps shows that it would fail. The bundler has the ability to not bundle the metacall after simulating it shows that it would fail and lead to a loss. Finally, this type of attack would be costly for a solver to perform as they'd pay the gas cost + surcharges on any intentionally failed solverOps.

There is also a level of trust that solvers have in an auctioneer when submitting their solverOps, as an auctioneer could steal all MEV anyway by analysing the opportunities in the metacall bundle. With that trust assumption in place, we can assume the auctioneer will not abuse the PerBlockLimit mechanism at solvers' expense.

Informational Severity Issues

I-01. Unnecessary variable: adjustedDeposits

Description: adjustedDeposits is never used in _adjustAccountingForFees(). We recommend just using the original t_ values on GasAccounting.sol line 440 instead and save some gas by decreasing the inputs and outputs of _adjustAccountingForFees().

Customer's Response:

Fixed during the refactor of the Gas Accounting system, in commit [fba1c9e]

I-02. Unnecessarily setting t_withdrawal on _credit()

Description: As t_withdrawal is never used after it is increased by amount at the end of the _credit() function, we recommend removing the line in question.

Customer's Response:

Fixed in commit [57b229b]

I-03. exPostBid terminology

Description: We recommend a more transparent terminology such as sortOnChain for example. As discussed with the team, this might not be possible at the moment due to backward compatibility issues, but it is something to keep in mind once backward compatibility can be overcome.

Customer's response:

Acknowledged, but exPostBids terminology left as is as some integrating partners are already familiar with the term.

I-04. postOpsCall hook is unnecessary

Description: After discussing with the team we arrived at the conclusion that postOpsCall could be combined with allocateValueCall, by making bool solverSuccessful and uint winningBid sent to the single hook and leaving it to the dapp to implement logic that is only run in the case of a winning solver in that hook.

Customer's response:

Fixed in commit [3269e9f]

I-05. _contribute() in ExecutionBase_.sol could be named better

Description: Change this function name to contributeToAtlas() for better clarity and to avoid confusion with the other contribute() in GasAccounting.sol

Customer's response:

Fixed in commit [cc0a4ff]

I-06.Unreachable but concerning DOS of a bundle

Description: This scenario is not technically possible but we believe it still should be shared as it could be relevant for future development of the protocol.

An attacker can flashloan a lot of native tokens (ETH, POL…) and call depositAndBond then call borrow and loop this depositAndBond and borrow as much as possible. Once the solver runs successfully finishes he returns the flashloan _amountSolverPays is very high (_withdrawals is super inflated not bound by the original ETH balance of the atlas contract) due to the fact that the borrowed ETH will be repaid from the bond (at the _assign function inside _settle). In the _assign there is a cast to uint112 from the amount (uint256) to be charged from the solver. This cast may cause a revert to the metacall and fail everything. This is possible because every time the attacker deposited, they increased their bond so at the end the reconciliation will pass.

Customer's response:

Acknowledged. Agreed that this attack should not be possible due to the loops needed to overflow in a uint112 (changed to a uint128 during the refactor of the Gas Accounting system).

Mitigation Review

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform

Atlas	https://github.com/FastLane-Labs/atlas	288f2ca7dcb8d0636b2e9f721e94cbfbfcefcaf2	EVM/Solidity 0.8

Project Overview

This document describes the Mitigation review of **Atlas** by manual code review findings. The work was undertaken from **March 20, 2025** to **April 10, 2025**.

The following contract list is included in our scope:

```
src/contracts/atlas/*
src/contracts/common/*
src/contracts/dapp/*
src/contracts/gasCalculator/*
src/contracts/helpers/Simulator.sol
src/contracts/helpers/Sorter.sol
src/contracts/libraries/*
src/contracts/types/*
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Please note that a few more formal rules are not included in this report, as they were proven with an unreleased version of the Certora Prover. Once those rules are proven on a released version of the Certora Prover, we will add them to the next version of this document.

certora

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	6	0	6
Medium	2	0	2
Low	6	1	5
Informational	1	0	1
Total	**15**	**1**	**14**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
H-01	Bundler can make msg.data bigger than it is, inflating gas costs	High	Fixed
H-02	_netRepayments is unfairly limited to _maxRefund	High	Fixed
H-03	_gL.remainingMaxGas isn't decreased correctly	High	Fixed
H-04	unreachedSolverGas is only decreased when signatures are valid	High	Fixed
H-05	Winning solver is unfairly punished if unreached solvers can't pay their bond	High	Fixed
H-06	solverOps are not verified if they are signed when calculating unreached calldata	High	Fixed
M-01	Winning solver pays for gas costs of handling failed solvers accounting	Medium	Fixed
M-02	Winning solver/bundler will pay for running the loop for each failed solver	Medium	Fixed

L-01	Surcharge rates can be changed mid metacall	Low	Acknowledged
L-02	unreachedCalldataValuePaid is used even when there are no winning solvers	Low	Fixed
L-03	_gasWaterMark initialized in incorrect place in function	Low	Fixed
L-04	SolverTxResult is emitted with incorrect variable	Low	Fixed
L-05	_gasUsed isn't capturing the correct gas left prior to solver fail accounting	Low	Fixed
L-06	Bundler can add duplicate solver ops, forcing them to fail and charging the solver	Low	Fixed

High Severity Issues

H-01 Bundler can make msg.data bigger than it is, inflating gas costs

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Atlas.sol	Fixed	

Description:

The bundler can make the calldata bigger than it is, because of the way the EVM interprets the offsets in the calldata we can make `msg.data.length` to be bigger than the sum of all it's parts. The bundler can do this without changing the `solverOps` or anything else, by just adding zeros in the middle, because the EVM use offsets in the calldata parsing the bundler can inflate only `msg.data.length` this charge goes to the winning solver who will pay all of his bond to do it by doing this we can drain the winning solver.

Recommendation:

Calculate only the calldata costs for `solverOps`, `userOp` and `dAppOp`, instead plainly using `msg.data`

Customer Response:

Fixed as per recommendation here.

H-02 _netRepayments is unfairly limited to _maxRefund		
Severity: **High**	Impact: **High**	Likelihood: **High**
Files: [GasAccounting.sol]()	Fixed	

Description:

`_netRepayments` represents the deposit made by the bundler and is currently capped by `_maxRefund` (80%) which is unfair to the bundler because this `_netRepayments` has the msg.value the bundler sent so capping this will cause the bundler to lose Eth.

Recommendation:

Add `_netRepayments` to `claimsPaidToBundler` at the end of the `else`, after the max refund logic as to not subject it to the limit.

Customer Response:

Fixed as per recommendation [here]().

H-03 _gL.remainingMaxGas isn't decreased correctly		
Severity: **High**	Impact: **High**	Likelihood: **High**
Files: Escrow.sol	Fixed	

Description:

In the findBid iteration loop we don't decrease the `_gL.remainingMaxGas` like here because the solvers are not failing and we do decrease and update `_gL.unreachedSolverGas` here inside `_validateSolverOpGasAndValue` function. This will cause the later solvers to have a higher result in this function solverGasLiability here which depending on where the solvers are at line affects the result of the bundler (still should not happen even if it is the bundlers fault) and more important if we pass that check we are affected here inside reconcile function which will affect the solver's bid in the bidFinding phase and this may affect the result where the solver could have had a better bid but was forced not to by this behaviour.

Recommendation:

Adjust the `_gL.unreachedSolverGas` so this will simulate the proper behaviour.

Customer Response:

Fixed as per the recommendation here.

H-04 unreachedSolverGas is only decreased when signatures are valid		
Severity: **High**	Impact: **High**	Likelihood: **High**
Files: Escrow.sol	Fixed	

Description:

In the knownBid path here we decrease the `unreachedSolverGas` but this happens only if the signature is valid. when the signature is not valid we just go to the `handleFail` and here we **always** decrease `remainingMaxGas` which means that is lets say the first couple of solvers just fail. in that case this will revert, the reason is that the `remainingMaxGas` will be smaller than `unreachedSolverGas` and the sub will underflow. This will cause the `metacall` to fail completely even if there are good solvers in the future. Also because this check for the `solverGasLiability` is wrong we can also (if we fail less solver so `remainingMaxGas` is still a bit bigger than `unreachedSolverGas` but less than it should) we don't really check that the solver has bond to pay we will execute and at the end there will be a deficit that the bundler will pay (no matter the result solver won or failed) this is bad for them.

Recommendation:

Decrease `unreachedSolverGas` even if the signature is invalid for both bid-finding execution and real execution.

Customer Response:

Fixed as per the recommendation here.

H-05 Winning solver is unfairly punished if unreached solvers can't pay their bond

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: Escrow.sol	Fixed	

Description:

If `exPostBids = false` then any uncreached solvers must pay for the gas costs of their calldata.

If they cannot cover the cost, the deficit is taken out of `unreachedCalldataValuePaid` which will be paid by the winning solver, effectively punishing him for being the winner. This can be even more impactful for the winning solver if the bundler and auctioneer collude by ordering the `solverOps` in such a way that solvers with large calldata costs will be placed after the winning solver, punishing him even harder and draining his bond potentially.

Recommendation:

Do not punish the winning solver as he has no control over which and how many solvers are after him. Instead, add the `_deficit` to the bundler's writeoffs.

Customer Response:

Fixed as per the recommendation here.

H-06 solverOps are not verified if they are signed when calculating unreached calldata		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: [GasAccounting.sol]()	Fixed	

Description:

In the case of `exPostBids = false`, any unreached solvers are charged for their calldata costs, by calling `_chargeUnreachedSolversForCalldata`. The calldata costs are calculated and then each unreached solver is charged for his calldata. The issue is that the actual `solverOp` is not verified, if `solverOp.from` signed it, because of this the bundler + auctioneer can add fake solver ops after the winning one, making the calldata massive and drain the bond of each targeted solver.

Recommendation:

Verify each unreached `solverOp` to make sure it was signed by the corresponding `solverOp.from`

Customer Response:

Fixed as per the recommendation here.

Medium Severity Issues

M-01 Winning solver pays for gas costs of handling failed solvers accounting		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: GasAccounting.sol	Fixed	

Description:

When a solver operation fails, it's handled through `_handleSolverFailAccounting`. We have two cases, if the solver op fails because it's the bundler's fault or because it's the solver's fault. The issue is that the gas costs of actually running the function are paid by the winning solver (if there is any), since none of these costs are charge neither to the bundler or the solver, unfairly punishing him for failed solver ops that he had nothing to do with. Considering the function can be run `solverOps.length - 1` amount of times, the gas costs of the winning solver can become quite large..

Recommendation:

Add the gas costs of running the function to `_gasUsed` and then add them to the bundler's writeoffs or to the solver fault failure gas.

Customer Response:

Fixed as per the recommendation here.

M-02 Winning solver/bundler will pay for running the loop for each failed solver		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Escrow.sol	Fixed	

Description:

When running `_bidFindingIteration` we loop through each valid solver op until we reach a winning solver. All the gas for running each `_executeSolverOperation` is either written off or added to the solver fault gas, but the actual gas costs of looping through each solver op isn't, which means at the end the winning solver (or bundler if all solver ops failed) will get charged, which is unfair to both parties.

Recommendation:

Remove the `_gasWaterMark` inside `_executeSolverOperation` and add a `_gasWaterMark` at the start of the loop which iterates over `_bidsAndIndicesLastIndex`.

Customer Response:

Fixed as per the recommendation here.

Low Severity Issues

L-01 Surcharge rates can be changed mid metacall		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: Storage.sol	Acknowledged	

Description:

Surcharge rates can be changed via `setAtlasSurchargeRate` during `metacall` if one of the solver ops is also `S_surchargeRecipient` which shouldn't be allowed. Consider adding `_checkIfUnlocked` inside `setAtlasSurchargeRate`.

Customer Response:

Acknowledged. As this is a permissioned operation, we believe there is very little risk of this happening as the Atlas surcharge recipient would never intentionally also be a solver. This part of the code is also refactored in the upcoming Atlas v1.6 upgrade, where this problem will be fully resolved.

L-02 unreachedCalldataValuePaid is used even when there are no winning solvers		
Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: [GasAccounting.sol]	Fixed	

Description:

`unreachedCalldataValuePaid` is unnecessarily added to `_bundlerCutBeforeLimit` when we have no winning solvers. If we have no winning solvers then `unreachedCalldataValuePaid` is always 0, so it's redundant to use it in this case.

Customer Response:

Fixed as per the recommendation here.

L-03 _gasWaterMark initialized in incorrect place in function		
Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: [Atlas.sol]()	Fixed	

Description:

Inside `_bidFindingIteration`, `_gasWaterMark` should be initialized at the start of the function to correctly capture the right amount of `_gasLeft` for the function.

Customer Response:

Fixed as per the recommendation [here]().

L-04 SolverTxResult is emitted with incorrect variable

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: Escrow.sol	Fixed	

Description:

`SolverTxResult` is emitted with `bidAmount` which is the previous bid. Use `_solverTracker.bidAmount` instead.

Customer Response:

Fixed as per the recommendation here.

L-05 _gasUsed isn't capturing the correct gas left prior to solver fail accounting		
Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: GasAccounting.sol	Fixed	

Description:

Inside `_handleSolverFailAccounting`, `_gL.remainingMaxGas` should be calculated before `_gasUsed` in order to correctly capture the gas left prior to handling solver fail accounting.

Customer Response:

Fixed as per the recommendation here.

L-06 Bundler can add duplicate solver ops, forcing them to fail and charging the solver		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: [Escrow.sol]	Fixed	

Description:

A malicious bundler can add duplicate solver ops to the `solverOps` array, because of this, each duplicate will fail because his `lastAccessedBlock == block.number` which results in a fault for the solver, not the bundler. We recommend implementing a way to track `solverOp.from` and if a duplicate appears, charge the bundler.

Customer Response:

Acknowledged. We assume solvers are sophisticated and will quickly realise if a malicious auctioneer/bundler collusion is duplicating their solver operations maliciously. Adding a duplicate check mechanism would increase gas cost, and there are other trust assumptions around the auctioneer, so for now we rely on those trust assumptions and save gas on the honest metacalls.

Informational Severity Issues

I-01. SolverTxResult should be before _handleSolverFailAccounting

Description: Consider emitting the event prior to calling `_handleSolverFailAccounting` so the party at fault pays for the event emission.

Customer Response:

Fixed as per the recommendation here.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora 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.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.