SPEARBIT

Fastlane Atlas Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Riley Holterhus, Lead Security Researcher

Blockdev, Security Researcher

Report prepared by: Lucas Goiriz

August 22, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Fastlane builds MEV-aware infrastructure and smart contracts to make DeFi useable and keep it decentralized. In particular, Atlas is a permissionless and modular smart contract framework for Execution Abstraction. It provides apps and frontends with an auction system in which Solvers compete to provide optimal solutions for user intents or MEV redistribution. A User Operation is collected by the app's frontend via the Atlas SDK and sent to a app-designated bundler, which combines it with Solver Operations into a single transaction.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of atlas according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 30 days in total, Fastlane engaged with Spearbit to review the atlas protocol. In this period of time a total of **190** issues were found.

Summary

Project Name	Fastlane
Repository	atlas
Commit	551eb4...861101
Type of Project	DeFi, MEV
Audit Timeline	Apr 1 to May 10
Two week fix period	Jul 1 - Jul 12

The Spearbit team reviewed Fastlane's atlas holistically on commit hash a6dd06836db92cddfb824ed88fa2e6953ce4a30e and determined that all issues were either fixed or appropriately acknowledged, and no new issues were identified

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	15	15	0
Medium Risk	25	24	1
Low Risk	35	31	4
Gas Optimizations	34	31	3
Informational	81	74	7
Total	**190**	**175**	**15**

5 Findings

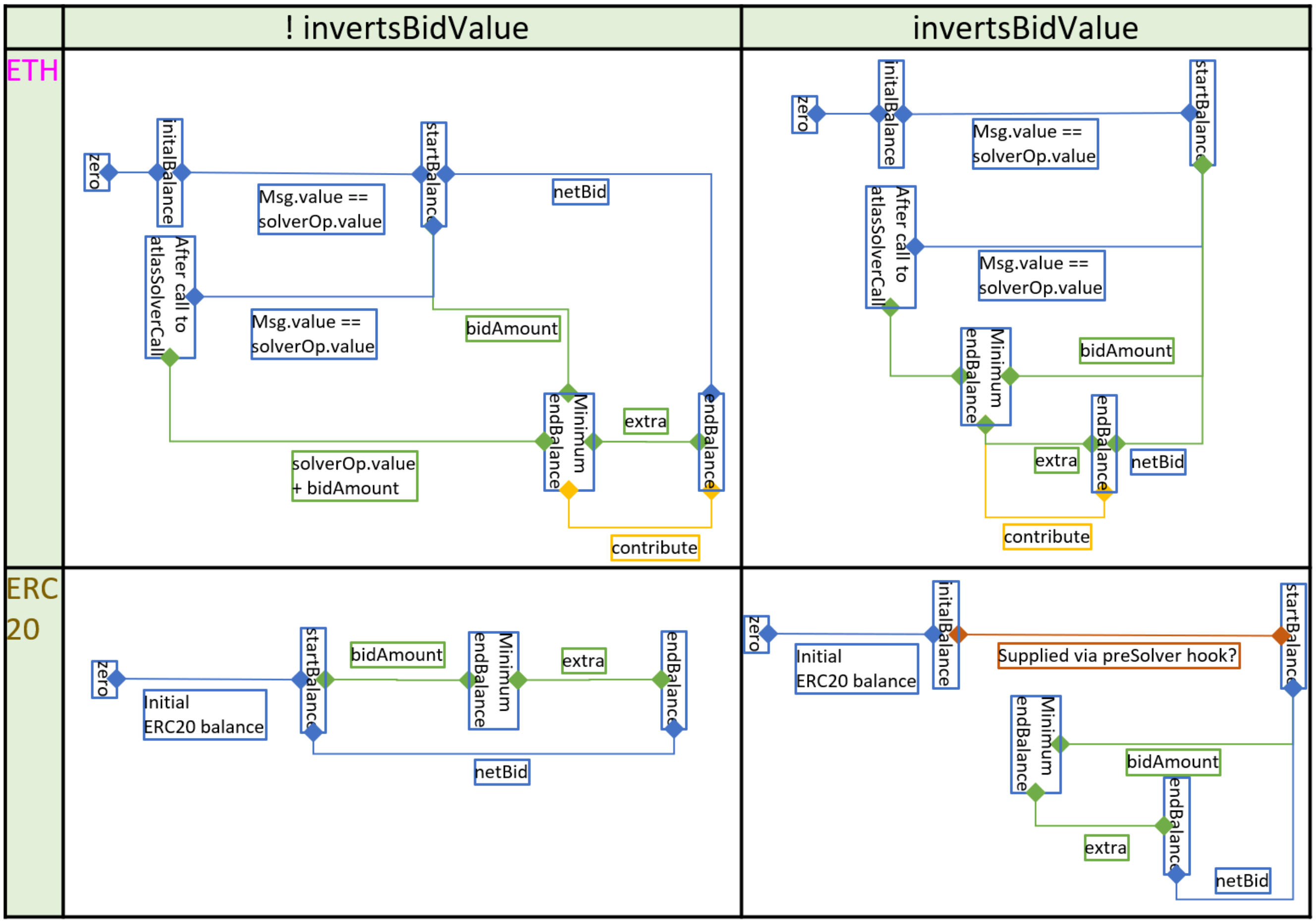
5.1 High Risk

5.1.1 Calculations in `solverMetaTryCatch()` non consistent

Severity: High Risk

Context: ExecutionEnvironment.sol#L142-L317

Description: The calculations in `solverMetaTryCatch()` non consistent. The drawings below show the different situations.



- `startBalance` **and** `endBalance` **calculations**

  The tables below show remarks concerning the `startBalance` and `endBalance` calculations. | ETH `startBalance` and `endBalance` | ! invertsBidValue | invertsBidValue| | ------------ | ----------------------| ------- | | bidFind==true | `endBalance` includes `solverOp.value` and `initial balance` so compare to `bidAmount` isn't ok | `startBalance == 0` so this fails: `netBid = startBalance - endBalance` | | bidFind==false | `endBalance` includes `solverOp.value` and `initial balance` so compare to `bidAmount` isn't ok| `startBalance == 0` so this fails: `startBalance - endBalance` |

ERC20`startBalance` and `endBalance`	! invertsBidValue	invertsBidValue
bidFind==true	Seems ok	Assumes sufficient ERC20 in EE
bidFind==false	Seems ok	Assumes sufficient ERC20 in EE No allowance to solver

When `invertsBidValue == false`, the `endBalance` is used, which includes the `solverOp.value` and the initial ETH balance. This is compared to `bidAmount` but that doesn't seem right.

The `startBalance` for `ETH` is set to 0 which leads to reverts when `invertsBidValue == true`, because the code tries to subtract from 0.

- **(ExtraEth)EndBalance calculations**

  Variable `endBalance` is reused for a second purpose. To make this clear, in the following text `(ExtraEth)EndBalance` is used.

  The tables below show remarks concerning the `(ExtraEth)EndBalance` that is `contribute`d to `Atlas`.

ETH (ExtraEth)EndBalance	! invertsBidValue	invertsBidValue
bidFind==true	`(ExtraEth)EndBalance` is always 0	`(ExtraEth)EndBalance` is always 0
bidFind==false	`(ExtraEth)EndBalance = endBalance - bidAmount`, however `endBalance` includes `solverOp.value` and `initial balance` should be something like: `(endBalance - startBalance ) - bidAmount`	`(ExtraEth)EndBalance = endBalance`, however `endBalance` includes `solverOp.value` and `initial balance` should be something like: `(startBalance - endBalance) - bidAmount`

ERC20 (ExtraEth)EndBalance	! invertsBidValue	invertsBidValue
bidFind==true	When extra ERC20 tokens `address(this).balance`, else 0	When extra ERC20 tokens `address(this).balance`, else 0
bidFind==false	address(this).balance	address(this).balance

For ETH and `bidFind==true`, the `(ExtraEth)EndBalance` is always 0, so the code could be simplified.

For ETH and `bidFind==false` the `(ExtraEth)EndBalance`calculations don't take into account that `endBalance` includes the `solverOp.value` and the initial ETH balance.

When `invertsBidValue == true` the startBalance for `ETH` is set to 0 which leads to reverts when `invertsBidValue == true`, because the code tries to subtract from 0.

With ERC20 and `invertsBidValue` case:

- The PreOps hook is supposed supply initial ERC20 tokens to the ExecutionEnvironment, there is no comment about this.
- The `solver` is supposed (as few as possible) ERC20 tokens from the ExecutionEnvironment, however no allowance is set for the `solver`.

For ERC20 the `(ExtraEth)EndBalance` is sometimes `address(this).balance` and sometimes 0, this doesn't seem right.

Recommendation: Check all situations carefully and adapt the code. Also taking into account this issue: "`solverMetaTryCatch()` assumes there is no pre-existing `ETH` in contract".

Prevent using variables for a second purpose, see "Reuse of variables is confusing".

Preferably make the code differences between the `_bidFind()== true` and the `_bidFind()== false` case as small as possible For example when `_bidFind() == true` then `_getBidAmount()` has supplied solverOp.bidAmount as input for bidAmount. So bidAmount could be used everywhere instead of solverOp.bidAmount.

Fastlane: Soved in PR 225.

Spearbit: Verified.

5.1.2 validControl / onlyAtlasEnvironment are not effective in delegatecall situation

Severity: High Risk

Context: DAppControl.sol#L20-L49, ExecutionBase.sol#L29-L47

Description: The checks validControl / onlyAtlasEnvironment are not effective in delegatecall situation:

If the ExecutionEnvironment does a delegatecall to a user contract, or inside any DappControl hook, any data can be provided at the end of the call parameters. Because it also has msg.sender == atlas a large number of calls are possible.

See for a proof of concept below.

Possible consequences:

- The user contract can also into all the hooks of the DappControl contract.
- The user contract can also into all other DappControl contracts.
  - This could also be done via a man in the middle attack: all delegate calls to an attacker DappControl are delegate called to the original DappControl contract. The original DappControl contract isn't aware of this.
- The user contract can also reenter into the ExecutionEnvironment, it can:
  - Adjust all configurations.
  - Execute all functions.
  - Call one (or more) solver contracts.
- The user contract can also call into other ExecutionEnvironments, but because it is a delegatecall and they all have same code that seems no problem.
- All DappControl hooks can call into the following (but the risk is limited because they already have access to the funds of the ExecutionEnvironment):
  - withdrawERC20()
  - factoryWithdrawERC20()
  - withdrawEther()
  - factoryWithdrawEther()

Here is a proof of concept:

```
contract UserContract {
    function callPreOpsCall(address control) public {
            console.log("in UserContract callPreOpsCall");
            UserOperation memory userOp;
            bytes memory data = abi.encodeWithSelector(DAppControl.preOpsCall.selector, userOp);
            bytes memory preOpsData = bytes.concat(data,
                abi.encodePacked(address(0),bool(false),bool(false),uint8(0),uint8(0),
                    uint16(type(uint16).max), // all states
                    uint24(0),bool(false),bool(false),uint8(2)),
                abi.encodePacked(address(0), address(control), uint32(0), bytes32(0))
            );
            (bool success, ) = address(control).delegatecall(preOpsData);
            console.logBool(success);
    }
}
```

```
modifier validControl() {
    if (CONTROL != _control()) revert AtlasErrors.InvalidControl();
    _;
}
modifier onlyAtlasEnvironment(ExecutionPhase phase, uint8 acceptableDepths) {
    _onlyAtlasEnvironment(phase, acceptableDepths);
    _;
}
function _onlyAtlasEnvironment(ExecutionPhase phase, uint8 acceptableDepths) internal view {
    if (address(this) == source) {
        revert AtlasErrors.MustBeDelegatecalled();
    }
    if (msg.sender != atlas) {
        revert AtlasErrors.OnlyAtlas();
    }
    if (uint16(1 << (EXECUTION_PHASE_OFFSET + uint16(phase))) & _lockState() == 0) {
        revert AtlasErrors.WrongPhase();
    }
    if (1 << _depth() & acceptableDepths == 0) {
        revert AtlasErrors.WrongDepth();
    }
}
```

Recommendation: Retrieve important variables and states directly from `Atlas`, for example `DappControl` contract, execution phase. Add reentrancy checks in the functions of `ExecutionEnvironment` or keep the `callIndex` / `callDepth` registration in `Atlas`. Call `solvers` from `Atlas`.

See for refactor approach: "Locking mechanism is complicated".

Fastlane: Added `onlyPhase` check in `DAppControl.sol` via PR 301. Also in PR 225 we changed `solver` calls to go directly from `Atlas` to the solver contract so that should take care of the solver safety concerns here.

Spearbit: Verified.

5.1.3 `reconcile()` creates `deposits` out of thin air

Severity: High Risk

Context: GasAccounting.sol#L73-L119

Description: A call to `reconcile()` with non-zero `maxApprovedGasSpend` increases `surplus` and `deposit`. Assuming `maxApprovedGasSpend` fits within the `bonded balance` of the `current solver`, this will later on be `_credited` to the solver. However there is no registration or backing of this `maxApprovedGasSpend`, so this creates a `bonded` balance out of thin air.

The comments say `This will be subtracted later`, but we couldn't find where this is done. The severity of this issue is increased by the following related issues:

- "`reconcile()` can be called by anyone".
- "Checks for `solverCalledBack` don't cover all situations".

```
function reconcile(address environment,address solverFrom,uint256 maxApprovedGasSpend) /*...*/ {
    // NOTE: approvedAmount is the amount of the solver's atlETH that the solver is allowing
    // to be used to cover what they owe.  This will be subtracted later - tx will revert here if there
↪   isn't
    // enough.
    // ...
    uint256 bondedBalance = uint256(accessData[solverFrom].bonded);
    if (maxApprovedGasSpend > bondedBalance) maxApprovedGasSpend = bondedBalance;
    // ...
    uint256 surplus = deposits + maxApprovedGasSpend + msg.value;
    // Add msg.value to solver's deposits
    if (msg.value > 0 || maxApprovedGasSpend > 0) deposits = surplus;  // deposits  is increased now
    // ...
}
```

Recommendation: Function `reconcile()` should register this reservation and subtract that later on.

Fastlane: Solved in PR 271. Before, we added `msg.value + maxApprovedGasSpend` to `deposits`. In PR, we only add `msg.value` to `deposits`, but still check that:

1) The solver has enough bonded `atlETH` to cover `maxApprovedGasSpend` and...
2) If `deposits + msg.value + maxApprovedGasSpend` will cover `claims + withdrawals - writeoffs`. If we pass that check we know we can assign those costs to the solver's `AtlETH` balance later in `_assign()`.

Spearbit: Verified.

5.1.4 Flag `_solverFulfilled` is unreliable

Severity: High Risk

Context: GasAccounting.sol#L29-L38, GasAccounting.sol#L73-L119, Storage.sol#L95-L100

Description: Function `reconcile()` sets the flag `_solverFulfilled` if sufficient funds are present. Later on `validateBalances()` trusts this flag and doesn't do any additional checks.

However after a call `reconcile()` it is still possible to do `_borrow()` and `_contribute()`, which change `withdrawals` and `deposits`. This could be done in the same hook that calls `reconcile()`.

```
function reconcile(/*...*/) /*...*/ {
    // ...
    uint256 deficit = claims + withdrawals;
    uint256 surplus = deposits + maxApprovedGasSpend + msg.value;
    // ...
    if (deficit > surplus) {
        // ...
        return deficit - surplus;
    }
    // CASE: Callback verified and solver duty fulfilled
    if (!calledBack || !fulfilled) {
        _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack | _solverFulfilled;
    }
    return 0;
}
function validateBalances() external view returns (bool calledBack, bool fulfilled) {
    (, calledBack, fulfilled) = solverLockData();
    if (!fulfilled) {
        uint256 _deposits = deposits;
        // Check if locked.
        if (_deposits != type(uint256).max) {
            fulfilled = deposits >= claims + withdrawals;
        }
    }
}
function solverLockData() public view returns (address currentSolver, bool calledBack, bool fulfilled) {
    uint256 solverLock = _solverLock;
    // ...
    fulfilled = solverLock & _solverFulfilled != 0;
}
```

Recommendation: In `validateBalances()` always check the end balances:

```
  function validateBalances() external view returns (bool calledBack, bool fulfilled) {
-     (, calledBack, fulfilled) = solverLockData();
-     if (!fulfilled) {
          uint256 _deposits = deposits;
          // Check if locked.
          if (_deposits != type(uint256).max) {
              fulfilled = deposits >= claims + withdrawals;
          }
-     }
  }
```

Remove the `_solverFulfilled` flag from `reconcile()`.

Fastlane: Fixed in PR 227 by only allowing `borrow()` to be called in `SolverOperation` phase or before.

Spearbit: Verified.

5.1.5 Nonce logic is skipped for smart contract wallets

Severity: High Risk

Context: AtlasVerification.sol#L532-L575

Description: If the `userOp.from` address is a smart contract, the `_verifyUser()` function makes a call to the user's `validateUserOp()` function. This call is expected to return a success `bool` value, which is instantly returned.

Since this return happens before the code reaches `_handleNonces()`, there is no nonce validation for smart contract wallets. Most smart contract validation functions (e.g. `validateUserOp()` in the case of ERC4337 or `isValidSignature()` in the case of ERC1271) do not manage nonces themselves, and rely on the caller for this. As a result, signatures can be replayed and nonces can be reused when the user is a smart contract wallet.

Also see the issue titled "Call to `validateUserOp()` won't work", which suggests replacing `validateUserOp()` with `isValidSignature()`.

Recommendation: Change the `_verifyUser()` control flow so that `_handleNonces()` is called even in the case of smart contract wallets.

Fastlane: Solved in PR 230.

Spearbit: Verified.

5.1.6 `_releaseSolverLock()` doesn't undo all the actions of `_trySolverLock()`

Severity: High Risk

Context: GasAccounting.sol#L208-L246, GasAccounting.sol#L129-L134

Description: The function `_releaseSolverLock()` doesn't undo all the actions of `_trySolverLock()`.

Function `_releaseSolverLock()` keeps `_solverLock` set to the (latest) `solver`, which means that in the `AllocateValue` hook and `PostOps` hook this value is still set. This allows `reconcile()` to still be called, even after it has been made authorized. See issue "`reconcile()` can be called by anyone".

Function `_releaseSolverLock()` doesn't undo the addition to `withdrawals`. This is good for the winning solver, because the ETH has been send to `solverMetaTryCatch()`. However if `solverMetaTryCatch()` reverts this is not good. The value of `withdrawals` will increase with every unsuccessful`solver` until eventually it is higher than the `address(this).balance`. After that the next `solvers` will fail because `_borrow()` will return `false`.

Also see issue "Check with `withdrawals` in `_borrow()` not correct ".

Furthermore `_releaseSolverLock()` isn't always called, see issues:

- "Solvers don't always reimburse the bundler".
- "Winning solver doesn't get gas costs `_assign()`ed".

Sometimes `_releaseSolverLock()` is called without `_trySolverLock()`:

- "`_releaseSolverLock()` can be run without `_trySolverLock()`".

Additionally function `_releaseSolverLock()` assigns used gas, which isn't shown in the function name.

```
function _trySolverLock(SolverOperation calldata solverOp) internal returns (bool valid) {
    if (_borrow(solverOp.value)) {
        _solverLock = uint256(uint160(solverOp.from));
        return true;
    } else {
        return false;
    }
}
function _releaseSolverLock(/*...*/) /*...*/ {
   // doesn't set  _solverLock
   // doesn't change withdrawals
}
function _borrow(uint256 amount) internal returns (bool valid) {
    // ...
    if (address(this).balance < amount + claims + withdrawals) return false;
    withdrawals += amount;
    return true;
}
```

Recommendation: Decrease `withdrawals` if `solverMetaTryCatch()` has `reverted`. This could be done by moving the `_borrow()` from `_trySolverLock()` to a `borrow()` inside `solverMetaTryCatch()`. In that case `{ value: solverOp.value }` should not be send to `solverMetaTryCatch()`.

After this change the values would be returned to their original value after a revert of `solverMetaTryCatch()`.

Also see issue "Check with `withdrawals` in `_borrow()` not correct ".

`_releaseSolverLock()` should preferably set `_solverLock` to `_UNLOCKED_UINT`. To save some gas this can also be done before the `AllocateValue` hook and `PostOps` hook, so for example at the end of function `_executeSolverOperation()`.

Consider changing the function name of `_releaseSolverLock()` to indicate it is also used to assign used gas.

Fastlane: Resolved in PR 271. There is no longer a `_trySolverLock()` function and instead the logic it used to hold (`_borrow()` the `solverOp.value` and if that succeeds, set `_solverLock` to the current solver) is now done directly in `solverCall()` before Atlas calls directly to the solver. If the solver call or the postSolver call fails, this `solverCall()` function fails in a try-catch style, reverting the effects of what used to be `_trySolverLock()`.

`_releaseSolverLock()` has been renamed to `_handleSolverAccounting()`

We still do not set `_solverLock` to `_UNLOCKED_UINT` in `_handleSolverAccounting()` (if a solver fails). The case of no successful solver is handled explicitly in `_settle()` and does not use the "stale" solver address in `_solverLock`.

Spearbit: Verified.

5.1.7 `solverMetaTryCatch()` assumes there is no pre-existing `ETH` in contract

Severity: High Risk

Context: ExecutionEnvironment.sol#L152

Description: `solverMetaTryCatch()` requires `ExecutionEnvironment's` balance to be the same as `solverOp.value`:

```
require(address(this).balance == solverOp.value, "ERR-CE05 IncorrectValue");
```

However, someone can frontrun this transaction and send some `ETH` to `ExecutionEnvironment` making its balance non-zero. This leads to the `solverMetaTryCatch()` call reverting, since the call is sent with an `ETH` amount equal to `solverOp.value`. This makes `address(this).balance > solverOp.value`. Since the error would be treated as `SolverOutcome.EVMError` in the `_solverOpWrapper()`, the solver would be forced to pay the gas costs for this revert.

Recommendation: Refactor the function as follows:

- Update ExecutionEnvironment.sol#L152 as:

```
- require(address(this).balance == solverOp.value, "ERR-CE05 IncorrectValue");
+ require(msg.value == solverOp.value, "ERR-CE05 IncorrectValue");
```

- Update `startBalance` initialization:

```
- startBalance = 0; // address(this).balance - solverOp.value;
+ startBalance = address(this).balance - msg.value;
```

Note: Additional changes are also necessary because `startBalance` isn't used everywhere. Also see issue "Calculations in `solverMetaTryCatch()` non consistent".

Fastlane: Resolved in PR 225 as `solver` calls no longer go through `EE` but now go directly from `Atlas` to the `solver` contract. So this balance check is no longer in the `EE`.

Spearbit: Additional improvement of `solverOp.value` check in PR 223. Verified.

5.1.8 Bid tokens aren't enforced to be the same

Severity: High Risk

Context: ExecutionEnvironment.sol#L322

Description: In Atlas, bid tokens are specified in multiple locations:

1. Within the `DAppConfig` (specifically from the `getBidFormat()` function).
2. Within each `SolverOperation`.

Currently, it's not enforced on-chain that these values are all consistent with each other. If the auctioneer or bundler includes different tokens in a transaction, the bid amount comparisons and the `allocateValueCall()` function would silently break, which could lead to unexpected results.

Recommendation: Add checks on-chain to ensure each `SolverOperation` contains the bid token that's specified in `DAppConfig`. The best location for this check is likely in the `verifySolverOp()` function.

Fastlane: Solved in PR 171.

Spearbit: Verified.

5.1.9 No slippage protection for UniswapV2 swaps

Severity: High Risk

Context: V2DAppControl.sol#L96-L109

Description: `amount0In` and `amount1In` values are dependent on `UniswapV2` pool's current token balance and on amount-out values. Someone can sandwich `Atlas` transaction to imbalance the pool leading to high amount-in value which is then transferred from user to the pool.

The attacker makes a profit through this sandwich and thus it's likely that all the token balance of the user is transferred to the pool.

Recommendation: Add slippage protection to the swap or at least make a comment about the lack of slippage protection.

Fastlane: Solved in PR 360 by making a comment.

Spearbit: Verified.

5.1.10 `bypassSignatoryApproval` skips important checks

Severity: High Risk

Context: AtlasVerification.sol#L372-L374

Description: In the initial `AtlasVerification` call, the `_verifyDApp()` function does various checks on the `dAppOp` argument. For example, see the following code snippet (with some comments removed for simplicity):

```
function _verifyDApp(
    DAppConfig memory dConfig,
    DAppOperation calldata dAppOp,
    address msgSender,
    bool bypassSignatoryApproval,
    bool isSimulation
)
    internal
    returns (bool, ValidCallsResult)
{

    bool bypassSignature = msgSender == dAppOp.from || (isSimulation && dAppOp.signature.length == 0);

    if (!bypassSignature && !_verifyDAppSignature(dAppOp)) {
        return (false, ValidCallsResult.DAppSignatureInvalid);
    }

    if (bypassSignatoryApproval) return (true, ValidCallsResult.Valid); // If bypass, return true after signature
        // verification

    if (dAppOp.bundler != address(0) && msgSender != dAppOp.bundler) {
        if (!signatories[keccak256(abi.encodePacked(dAppOp.control, msgSender))]) {
            bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0);
            if (!isSimulation) {
                return (false, ValidCallsResult.InvalidBundler);
            }
        }
    }

    if (!signatories[keccak256(abi.encodePacked(dAppOp.control, dAppOp.from))]) {
        bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0);
        if (!bypassSignatoryCheck) {
            return (false, ValidCallsResult.DAppSignatureInvalid);
        }
    }

    if (dAppOp.control != dConfig.to) {
        return (false, ValidCallsResult.InvalidControl);
    }

    if (dAppOp.from == address(0) && isSimulation) {
        return (true, ValidCallsResult.Valid);
    }

    if (!_handleNonces(dAppOp.from, dAppOp.nonce, !dConfig.callConfig.needsSequencedDAppNonces(), isSimulation)) {
        return (false, ValidCallsResult.InvalidDAppNonce);
    }

    return (true, ValidCallsResult.Valid);
}
```

There are six main checks in this function:

1. A check that `dAppOp.from` has authorized the transaction (either as `msgSender` or through a signature).
2. A check that the `msgSender` is authorized to act as a bundler.
3. A check that the `dAppOp.from` is authorized to act as the auctioneer.
4. A check that ensures `dAppOp.control == dConfig.to` .
5. A simulation check for `dAppOp.from == address(0)`.
6. A nonce check (with associated logic that will invalidate the used nonce).

This function also includes a `bypassSignatoryApproval` boolean, which will skip checks 2-6 if `true`. However, some of these checks should not be skipped, for example, the `dAppOp.control` check (number 4) and the nonce logic (number 6) seem important to execute regardless of the value of `bypassSignatoryApproval`. With the nonce check specifically, this behavior would sometimes allow `dAppOp` nonces to be reused or executed in an unexpected order.

Recommendation: Consider moving the important checks to before the `bypassSignatoryApproval` early return. In particular, the `dAppOp.control == dConfig.to` check and the call to `_handleNonces()` should be moved so that they are always executed.

Fastlane: Solved by PR 177.

Spearbit: Verified.

5.1.11 Incorrect indexing for bid sorting algorithm

Severity: High Risk

Context: Atlas.sol#L265-L273

Description: When an Atlas `CallConfig` specifies `exPostBids == true`, all `solverOps` are simulated on-chain to determine their theoretical bid amount. The `solverOps` are then sorted and executed in order until a `solverOp` succeeds. The sorting of the bids is facilitated through the following code:

```
uint256[] memory sortedOps = new uint256[](solverOps.length);
uint256[] memory bidAmounts = new uint256[](solverOps.length);
uint256 j;
uint256 bidPlaceholder;

for (uint256 i; i < solverOps.length; i++) {
    bidPlaceholder = _getBidAmount(dConfig, userOp, solverOps[i], returnData, key);

    if (bidPlaceholder == 0) {
        unchecked {
            ++j;
        }
        continue;
    } else {
        bidAmounts[i] = bidPlaceholder;

        for (uint256 k = i - j + 1; k > 0; k--) {
            if (bidPlaceholder > bidAmounts[sortedOps[k - 1]]) {
                sortedOps[k] = sortedOps[k - 1];
                sortedOps[k - 1] = i;
            } else {
                sortedOps[k] = i;
                break;
            }
        }
    }
}
```

Notice that the inner `for` loop starts with the index `k = i - j + 1`. Since it's possible that `j` always remains at `0` (i.e. if all bid simulations succeed), this index may be out-of-bounds for the `sortedOps` array, which will cause an unintended revert. This indexing can also potentially leave the zeroth index unset, which can later lead to duplicate attempts of the first `solverOp`.

Here is a proof of concept to show the issue:

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity 0.8.25;
import "hardhat/console.sol";

contract test {
    constructor() {
        uint ol = 2;
        uint256[] memory _getBidAmount = new uint256[](ol);
        _getBidAmount[0] = 6; // works if one of these values is 0
        _getBidAmount[1] = 6;
        uint256[] memory sortedOps = new uint256[](ol);
        uint256[] memory bidAmounts = new uint256[](ol);
        uint256 j;
        uint256 bidPlaceholder;

        for (uint256 i; i < ol; i++) {
            bidPlaceholder = _getBidAmount[i];
            if (bidPlaceholder == 0) {
                unchecked { ++j;}
                continue;
            } else {
                bidAmounts[i] = bidPlaceholder;
                for (uint256 k = i - j + 1; k > 0; k--) {
                    if (bidPlaceholder > bidAmounts[sortedOps[k - 1]]) {
                        sortedOps[k] = sortedOps[k - 1];
                        sortedOps[k - 1] = i;
                    } else {
                        sortedOps[k] = i;
                        break;
                    }
                }
            }
        }
        uint total = ol - j;
        console.log("total",total);
        for (uint256 i; i < total; i++) {
            console.log(i,sortedOps[i],bidAmounts[sortedOps[i]]);
        }
    }
}
```

Recommendation: Rework the indexing of this sorting algorithm. The following code snippet is one possible implementation of the inner `for` loop:

```
// `k` starts at the left-most unfilled location
uint256 k = i - j;
while (k > 0 && bidPlaceholder > bidAmounts[sortedOps[k - 1]]) {
    sortedOps[k] = sortedOps[k - 1];
    k--;
}
sortedOps[k] = i;
```

Alternatively, consider reworking a larger part of this code to make the on-chain sorting easier to understand.

Fastlane: Fixed in PR 154.

Spearbit: Verified.

5.1.12 `userOp` validation is skipped in simulation mode for smart contract user accounts

Severity: High Risk

Context: AtlasVerification.sol#L551

Description: It can happen that user has not allowed this userOp (ie, returning `validateUserOp()` returns `false`), but during simulation it's falsely believed that the user has allowed it since `isSimulation` is true:

```
bool validSmartWallet =
    IAccount(userOp.from).validateUserOp{ gas: 30_000 }(userOp, _getProofHash(userOp), 0) == 0;
return (isSimulation || validSmartWallet);
```

If this `userOp` goes onchain, it leads to a revert wasting gas for the bundler.

Recommendation: Check that smart contract user account has approved `userOp` even in simulation mode:

```
- return (isSimulation || validSmartWallet);
+ return (validSmartWallet);
```

Fastlane: Solved by PR194 and PR 250.

Spearbit: Verified.

5.1.13 `ExecutionEnvironment` deployment can be incorrectly skipped

Severity: High Risk

Context: Factory.sol#L56, Factory.sol#L106

Description: `ExecutionEnvironment` is deployed iff the address `executionEnvironment`, at which it's going to be deployed, has no code. The is checked by ensuring `executionEnvironment.codehash` is `0`. However, someone can frontrun this transaction by sending some `ETH` to this address. Now `codehash` returns a non-zero hash and `executionEnvironment` is never deployed.

An address which doesn't have code but has any non-zero ether balance returns `keccak256("")` as its codehash. This is as per the following EIPs:

- From `https://eips.ethereum.org/EIPS/eip-161`:

  > An account is considered empty when it has no code and zero nonce and zero balance.

- From `https://eips.ethereum.org/EIPS/eip-1052`:

  > In case the account does not exist or is empty (as defined by EIP-161) 0 is pushed to the stack.
  >
  > In case the account does not have code the `keccak256` hash of empty data (i.e. `c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`) is pushed to the stack.

Recommendation: Check for `executionEnvironment.code.length` instead.

Fastlane: Solved in PR 155 by checking `code.length`.

Spearbit: Verified.

5.1.14 Wrong ERC20 token transferred

Severity: High Risk

Context: SwapIntent.sol#L129

Description: `tokenUserBuys` ERC20 token is transferred here when it's meant to be `auctionBaseCurrency` since the balance amount transferred is corresponding to `auctionBaseCurrency`:

```
if (auctionBaseCurrencyBalance > 0) {
    ERC20(swapIntent.tokenUserBuys).safeTransfer(user, auctionBaseCurrencyBalance);
}
```

Recommendation:

```
- ERC20(swapIntent.tokenUserBuys).safeTransfer(user, auctionBaseCurrencyBalance);
+ ERC20(swapIntent.auctionBaseCurrency).safeTransfer(user, auctionBaseCurrencyBalance);
```

Fastlane: Solved in PR 165.

Spearbit: Verified.

5.1.15 `Borrow()`s after `validateBalances()`

Severity: High Risk

Context: GasAccounting.sol#L29-L55, GasAccounting.sol#L254-L298

Description: The function `borrow()` can still be called in the `AllocateValue` and `PostOps` phases. As this is after `validateBalances()` the `solver` has to pay for this in `_settle()`. However the `solver` is no longer in control and would be griefed this way.

Another risk is highlighted in the issue "Circumvent `AtlETH` unbonding period".

Recommendation: The most logical would be to restrict access to `borrow()` in these phases. However the logic from `SafetyBits` doesn't work in `Atlas` / `borrow()` because the variable `key` isn't accessible. To solve this, the `ExecutionPhase` should be kept at the `Atlas` level.

See issue "Locking mechanism is complicated".

Fastlane: Fixed in PR 227 by only allowing `borrow()` to be called in `SolverOperation` phase or before.

Spearbit: Verified.

5.2 Medium Risk

5.2.1 Simulation success may not guarantee on-chain success

Severity: Medium Risk

Context: Global scope

Description: Before a bundler submits an `Atlas` transaction on-chain, they will simulate the transaction off-chain to ensure it succeeds. This will likely be done using the `Simulator` helper contract, or if the suggestion from the issue "Simulation code path can be kept off-chain" is taken, with some other off-chain method. Regardless of the approach taken, this is an important step of the `Atlas` process, since bundlers will waste `ETH` on gas costs if a transaction reverts.

It's important to note that there are situations where off-chain simulation success does not guarantee on-chain success. This is a known concern in systems like ERC4337, which handles this problem with a specific simulation procedure described in ERC7562. For example, this specification disallows certain sections of code from using opcodes that can easily trick simulation (e.g. `TIMESTAMP` and `COINBASE`).

In `Atlas`, there are a few locations where this may be a similar concern. This includes:

- If `userOp.from` is a smart contract, the `_verifyUser()` function calls the contract to verify the user's signature. If there aren't any restrictions on the contract's implementation, it may contain malicious logic designed to revert on-chain and waste the bundler's ETH.
- If all `solverOps` fail, either the `UserNotFulfilled()` revert happens, or the bundler is treated as the winning solver, and in either case, the bundler is not reimbursed for all gas fees. Since `solverOps` can contain arbitrary logic, they may revert on-chain even after a successful off-chain simulation.
- If any of the `preOpsWrapper()`, `userWrapper()`, or `postOpsWrapper()` functions revert, then the entire `execute()` call reverts and the bundler will not be reimbursed any gas. This implies that a `userOp` or `DAppControl` could grief the `bundler` if it's designed to trick simulation.

Recommendation: Consider if any of the above situations require mitigations for `bundler` protection. Depending on how concerning an on-chain revert is, this may be partially solved with an off-chain allowlist/reputation system that tracks each party's success rate.

Fastlane: Partly solved in PR 362 by adding a template for additional checks.

Spearbit: Verified. After discussion with the team, it was determined that this attack vector is very difficult to completely mitigate. Bundlers can reduce their risk of wasting gas by being careful with the DApps/users they interact with.

5.2.2 No quorum requirements for `transmit()` function

Severity: Medium Risk

Context: ChainlinkAtlasWrapper.sol#L77-L124, ChainlinkDAppControl.sol#L136-L162

Description: In order to `transmit()` a new oracle update to the `ChainlinkAtlasWrapper`, a transmitter provides the signatures of other whitelisted signers that are attesting to the oracle update.

However, neither of the `transmit()` or `verifyTransmitSigners()` functions verify the amount of signatures submitted, and providing an empty array of signatures will technically succeed. This behavior changes the trust assumptions of the `ChainlinkDAppControl` since a transmitter has full control of each oracle update.

Recommendation: Add a quorum check into the `transmit()` flow. One way to implement this is to check against the `verificationVars[baseChainlinkFeed].signers.length` value in the `verifyTransmitSigners()` function. For example, the following code would require that *all* signers be submitted by the transmitter:

```
  function verifyTransmitSigners(
      address baseChainlinkFeed,
      bytes calldata report,
      bytes32[] calldata rs,
      bytes32[] calldata ss,
      bytes32 rawVs
  )
      external
      view
      returns (bool verified)
  {
      bool[] memory signed = new bool[](MAX_NUM_ORACLES);
      bytes32 reportHash = keccak256(report);
      Oracle memory currentOracle;
+     require(rs.length == verificationVars[baseChainlinkFeed].signers.length);

      for (uint256 i = 0; i < rs.length; ++i) {
          address signer = ecrecover(reportHash, uint8(rawVs[i]) + 27, rs[i], ss[i]);
          currentOracle = verificationVars[baseChainlinkFeed].oracles[signer];

          // Signer must be pre-approved and only 1 observation per signer
          if (currentOracle.role != Role.Signer || signed[currentOracle.index]) {
              return false;
          }
          signed[currentOracle.index] = true;
      }
      return true;
  }
```

Fastlane: Solved in PR 184.

Spearbit: Verified.

5.2.3 getBidValue() is not always used

Severity: Medium Risk

Context: Sorter.sol#L118, Atlas.sol#L325-L328, Escrow.sol#L355, SwapIntent.sol#L243-L245

Description: Sorter uses getBidValue() to sort the bids. However Atlas / _bidFindingIteration() and escrow / _getBidAmount() don't do that and use solverOp.bidAmount directly.

In the example code these values are the same because the following function is used. However in the general case they might be different.

```
function getBidValue(SolverOperation calldata solverOp) public pure override returns (uint256) {
    return solverOp.bidAmount;
}
```

Recommendation: Also use getBidValue() in Atlas / _bidFindingIteration() and escrow / _getBidAmount(), or if the use is limited remove it from Sorter to be consistent.

Fastlane: Solved in PR 372 by removing the call from the Sorter.

Spearbit: Verified.

5.2.4 Hashes don't depend on the DAppControl config state

Severity: Medium Risk

Context: AtlasVerification.sol#L273-L291, AtlasVerification.sol#L481-L497, AtlasVerification.sol#L580-L597

Description: The control address contributes to the hash calculation of the UserOperation, solverOp, and DAppOperation structs. This means that when a party provides a signature for verification, they specify exactly which DAppControl address they're interacting with. However, it should be noted that it's possible for a DAppControl to change its behavior while remaining at the same address.

For example, a DAppControl can be programmed to return different a CallConfig value on separate calls. While this would change the underlying ExecutionEnvironment address used, the signatures for each UserOperation, solverOp, and DAppOperation would remain valid, which can add unexpected trust assumptions. For instance, a solverOp is not replayable as long as the corresponding solverOp.userOpHash only appears on-chain once. However, if a DAppControl flips its userNoncesSequenced boolean, the same userOp nonce can be used in two different contexts, which might allow the solverOp to be replayed.

Recommendation: Consider incorporating each party's expected CallConfig value into the struct they provide, and add the value into the hash calculation. This allows for stronger protections against a DAppControl changing its behavior unexpectedly.

Fastlane: Solved in PR 173.

Spearbit: Verified.

5.2.5 Solvers can be unfairly forced into gas refunds

Severity: Medium Risk

Context: AtlasVerification.sol#L207, Escrow.sol#L256-L267

Description: The EscrowBits library defines the circumstances when a solver is required to refund the bundler for gas costs. For example, the SolverOutcome.BidNotPaid flag is part of the _FULL_REFUND value, which means a solver needs to reimburse their gas usage if their solverOp fails due to an insufficient bid.

Since these gas costs are forced on the solver, it's important that each error leading to reimbursement is actually something the solver is at fault for. This does not always seem to be the case currently, including the following:

- The SolverOutcome.DeadlinePassed flag is part of the _PARTIAL_REFUND value, although the timestamp when a solverOp is included on-chain is something the bundler controls. So, solvers are unnecessarily punished if the bundler includes their solverOp late.

- The SolverOutcome.GasPriceOverCap flag is part of the _PARTIAL_REFUND value, although the tx.gasprice is something in control of the bundler. So, if the bundler specifies an unreasonably large priority fee, the solver is ultimately punished for not accepting the price. This is also relevant to the SolverOutcome.GasPriceBelowUsers flag when allowsTrustedOpHash() == true, as the solver does not know the userOp.maxFeePerGas ahead of time.
- The SolverOutcome.PreSolverFailed flag is part of the _PARTIAL_REFUND value, although the failure of the preSolverCall() may be due to the DAppControl and not the solver. The SolverOutcome.EVMError similarly punishes the solver but may be caused by an error during the abi.decode() on the preSolverCall() return value (note: this is suggested to be removed in the issue titled "preSolverCall() can revert instead of returning false"). These are contrary to the fact that the SolverOutcome.AlteredControl error can also be caused by unexpected preSolverCall() behavior, but is a part of the _NO_REFUND category.
- The SolverOutcome.PerBlockLimit flag is part of the _PARTIAL_REFUND value, but solvers may not be able to prevent multiple of their solverOps executing in the same block. For example, if a solver interacts with a DAppControl with allowsTrustedOpHash() == true, they may not know exactly when their transactions will be executed on-chain, and may accidentally be included twice in a block.

Recommendation: For each error that is outside of the solver's control, move the corresponding error flag to the _NO_REFUND value in the EscrowBits library.

Fastlane: Addressed the relevant concerns in PR 271.

Spearbit: Verified. After a discussion about which errors are relevant to fix, it was determined:

- The SolverOutcome.DeadlinePassed and SolverOutcome.GasPriceOverCap errors are not reachable if allowsTrustedOpHash == false and the solver sets their deadline and gas price less strictly than the user. A new SolverOutcome.GasPriceBelowUsersAlt error has been added so that solvers are not blamed in the allowsTrustedOpHash == true case, which solves the problem mentioned in the issue.
- The SolverOutcome.PreSolverFailed errors are avoidable by trusting and understanding each DAppControl contract. Also the SolverOutcome.AlteredControl error can't be caused by a DAppControl anymore, which solves the inconsistency mentioned in the issue.
- The SolverOutcome.PerBlockLimit error would have other tradeoffs if the costs were forced on the bundler. There are methods that the solver can use to avoid this error (e.g. using multiple accounts), so this issue is acknowledged in its current form.

5.2.6 _bidFindingIteration doesn't reset key.callIndex

Severity: Medium Risk

Context: Atlas.sol#L237-L295, Escrow.sol#L320-L387, SafetyBits.sol#L87-L94, LockTypes.sol#L4-L18

Description: _bidFindingIteration() sends key as a memory parameter to _getBidAmount(), which means it is sent by reference. _getBidAmount() increments key.callIndex via holdSolverLock().

Values of key that are updated in _bidFindingIteration() can be used in the following loop iteration. If that would not work, then key.callIndex would be the same every time. Also see: "Passsing of key can be simplified".

After the last loop the value of key.callIndex == solverOps.length. Then the second loop with _executeSolverOperation() starts, which continues to use key.callIndex. So key.callIndex could end up to be 2x solverOps.length, depending on the winning solver.

However considering issue "callIndex incremented twice" in _executeSolverOperation() : key.callIndex could end up to be 3× solverOps.length, depending on the winning solver. Because callIndex is of type uint8, only 256/3 == 83 solvers can be supported, which is a lot less than MAX_SOLVERS (253).

```
struct EscrowKey {
    // ...
    uint8 callIndex;
    // ...
}
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    for (uint256 i; i < solverOps.length; i++) {
        bidPlaceholder = _getBidAmount(dConfig, userOp, solverOps[i], returnData, key);
        // ...
    }
    // key.callIndex == solverOps.length
    for (uint256 i; i < j; i++) {
        // ...
        (auctionWon, key) = _executeSolverOperation(..., key); // continues to use key
        // ...
    }
}
function _getBidAmount(..., EscrowKey memory key) /*...*/ {
    // ...
    data = abi.encodePacked(data, key.holdSolverLock(solverOp.solver).pack()); // increment callIndex
    // ...
}
function holdSolverLock(EscrowKey memory self, address nextSolver) internal pure returns (EscrowKey
↪   memory) {
    // ...
    ++self.callIndex;
    // ...
}
```

Here is a proof of concept that shows the issue:

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity 0.8.25;
import "hardhat/console.sol";

struct EscrowKey {
    uint8 callIndex;
}
library SafetyBits {
    function holdSolverLock(EscrowKey memory self) internal pure returns (EscrowKey memory) {
        ++self.callIndex;
        return self;
    }
}
contract test {
    using SafetyBits for EscrowKey;
    function _bidFindingIteration(EscrowKey memory key) public {
        uint ol = 10;
        for (uint256 i; i < ol; i++) {
            _getBidAmount(key);
        }
        console.log("key.callIndex=",key.callIndex); // 10
    }
    function _getBidAmount(EscrowKey memory key) internal {
        key.holdSolverLock();
    }
    constructor() {
        EscrowKey memory key;
        _bidFindingIteration(key);
    }
}
```

Recommendation: Reset `key.callIndex` between the for loops. Here is a straightforward example to show the idea:

```
  function _bidFindingIteration(...) ... {
+     uint8 saveCallIndex = key.callIndex;
      // ...
      for (uint256 i; i < solverOps.length; i++) {
          bidPlaceholder = _getBidAmount(dConfig, userOp, solverOps[i], returnData, key);
          // ...
      }
+     key.callIndex = saveCallIndex;
      for (uint256 i; i < j; i++) {
          // ...
          (auctionWon, key) = _executeSolverOperation(..., key);
          // ...
      }
  }
```

Fastlane: Solved by a different implementation in PR 225.

Spearbit: Verified.

5.2.7 `atlasSolverCall()` doesn't check caller

Severity: Medium Risk

Context: SolverBase.sol#L29-L61

Description: Function `atlasSolverCall()` doesn't check if its called from/via `Altas`. It does check `sender`, but this is a user supplied variable so has no guarantees. Without checks the code might potentially be abused.

```
function atlasSolverCall( address sender, /*...*/ ) /*...*/ safetyFirst(sender) /*...*/ {
    // ...
}
modifier safetyFirst(address sender) {
    require(sender == _owner, "INVALID CALLER");
    // ...
}
```

Recommendation: The most straightforward way would be to call `atlasSolverCall()` directly from `Atlas`. See "Locking mechanism is complicated".

Fastlane: Solved in PR 225 as `solver` calls no longer go through the Execution Environment but now go directly from `Atlas` to the `solver` contract, where `msg.sender == _atlas` is checked in the example implementation.

Spearbit: Verified.

5.2.8 Circumvent `AtlETH` unbonding period

Severity: Medium Risk

Context: GasAccounting.sol#L143-L185

Description: The function `_assign()` allows the usage of ETH that is bonded. This is what it is designed for. Here is an approach to abuse this:

- Assume one party combines all roles: user, auctioneer, bundler, solver and DappControl.
- The party borrows ETH after `validateBalances()`, see issue "`Borrow()`s after `validateBalances()`".
- Assume the borrowed amount is less than the bonded balance of the party.
- With `_settle()`, the borrowed amount subtracted from the bonded balance of the party.
- The party still has the borrowed amount.

So effectively ETH is freed while it was bonded, without have to wait for the `AtlETH` unbond period.

```
function _assign(address owner, uint256 amount, bool solverWon, bool bidFind) internal returns (bool
↪   isDeficit) {
    // ...
    EscrowAccountAccessData memory aData = accessData[owner];
    if (aData.bonded < amt) {
        // ...
    } else {
        aData.bonded -= amt;
    }
    accessData[owner] = aData;
    // ...
}
```

Recommendation: See the solution for "`Borrow()`s after `validateBalances()`".

Fastlane: Fixed in PR 227 by only allowing `borrow()` to be called in `SolverOperation` phase or before.

Spearbit: Verified.

5.2.9 Check with `withdrawals` in `_borrow()` is incorrect

Severity: Medium Risk

Context: GasAccounting.sol#L48-L55, GasAccounting.sol#L129-L134

Description: The check with `withdrawals` in function `_borrow()` doesn't seem correct because `balance` is decreased with `safeTransferETH`, when `withdrawals` is increased so it counts double.

This example shows the issue:

- Assume `claims` and the initial value of `withdrawals` are neglectible.
- Assume `atlas` contains 100 ETH.
- Try to `borrow` 75 ETH:
  - `(address(this).balance < amount + claims + withdrawals)` ⇒ `100 ETH < 70 ETH + 0 + 0` ⇒ ok to borrow.
  - After this. `withdrawals == 75 ETH` and `balance == 25 ETH`.
- Now try to borrow an extra 10 ETH.
  - `(address(this).balance < amount + claims + withdrawals)` ⇒ `25 ETH < 10 ETH + 0 + 75 ETH` ⇒ not ok to borrow.

So you can't borrow the extra 10 ETH although `atlas` still has enough ETH. However if you directly borrow 85 ETH then there is no problem.

Also see issue "`_releaseSolverLock()` doesn't undo all the actions of `_trySolverLock()`" for another issue with the check in `_borrow()`.

```solidity
function borrow(uint256 amount) external payable {
    // ...
    if (_borrow(amount)) {
        SafeTransferLib.safeTransferETH(msg.sender, amount);
    } else {
        revert InsufficientAtlETHBalance(address(this).balance, amount);
    }
}
function _borrow(uint256 amount) internal returns (bool valid) {
    if (amount == 0) return true;
    if (address(this).balance < amount + claims + withdrawals) return false;
    withdrawals += amount;
    return true;
}
```

Recommendation: Consider changing the code to:

```diff
- if (address(this).balance < amount + claims + withdrawals) return false;
+ if (address(this).balance < amount + claims) return false;
```

Fastlane: Solved in PR 234.

Spearbit: Verified.

5.2.10 Checks for solverCalledBack don't cover all situations

Severity: Medium Risk

Context: ExecutionEnvironment.sol#L142-L317, GasAccounting.sol#L73-L119, Storage.sol#L95-L100

Description: Function solverMetaTryCatch() checks reconcile() isn't called by the PreSolver and that it is called by the PostSolver. However these checks don't cover all situations:

- If needsPreSolver() == false then the first check isn't done.
- If needsSolverPostCall() == false then the second check isn't done.

```
function solverMetaTryCatch(/*...*/) /*...*/ {
    // ...
    if (config.needsPreSolver()) {
        // call PreSolver
        // ...
        (, success,) = IEscrow(atlas).solverLockData();  // check reconcile() has been called
        if (success) revert AtlasErrors.InvalidEntry();
    }
    // call atlasSolverCall
    if (config.needsSolverPostCall()) {
        // Verify that the solver contract hit the callback before handing over to PostSolver hook
        (, success,) = IEscrow(atlas).solverLockData(); // check reconcile() has been called
        if (!success) revert AtlasErrors.CallbackNotCalled();
        // call postSolverCall
    }
    // ...
}
function reconcile(/*...*/) // ...
    // ...
    _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack;
   // ...
}
function solverLockData() public view returns (address currentSolver, bool calledBack, bool fulfilled) {
    uint256 solverLock = _solverLock;
    // ...
    calledBack = solverLock & _solverCalledBack != 0;
    // ...
}
```

Recommendation: Consider removing these checks here and refactor the code. Then _solverLock doesn't have to be set in reconcile(). Note _solverLock is makes sure that reconcile() is only called once, depending on the refactor an alternative solution might be required.

Access to reconcile() could be restricted to specific phases. Also see issue "Locking mechanism is complicated".

Fastlane: Solved in PR 227 as reconcile() is now restricted to just the SolverOperations phase. Additionally, the calledBack and fulfilled checks are now done after the preSolver, solver, and postSolver hooks, and are not dependent on any combination of pre/post Solver hooks being enabled.

Spearbit: Verified.

5.2.11 ChainlinkAtlasWrapper may break protocol integrations

Severity: Medium Risk

Context: ChainlinkAtlasWrapper.sol

Description: The ChainlinkAtlasWrapper is intended to be used by integrating protocols as a replacement for the original BASE_FEED Chainlink contract. There are currently two things that would make this integration difficult:

1. The latestRoundData() function does not maintain the behavior from the original Chainlink oracle. This function always returns the current roundId, startedAt, and answeredInRound from the BASE_FEED. This means that an oracle update in the ChainlinkAtlasWrapper will change the overall answer, but will not change the corresponding roundId. This implies that one roundId can have multiple answers, which is not possible in the original Chainlink contracts.
2. There are some functions missing in the ChainlinkAtlasWrapper. For example, the decimals() and getRoundData() functions are commonly used by protocols, but do not exist in the ChainlinkAtlasWrapper.

Recommendation: Consider if the roundId system from the BASE_FEED Chainlink contracts can be adopted. This may be difficult since the BASE_FEED does not know about the ChainlinkAtlasWrapper, and thus might have colliding ids.

Also, consider adding more Chainlink functionality to the ChainlinkAtlasWrapper. The commonly used oracle interface is the AggregatorV2V3Interface below:

```
interface AggregatorInterface {
  function latestAnswer() external view returns (int256);
  function latestTimestamp() external view returns (uint256);
  function latestRound() external view returns (uint256);
  function getAnswer(uint256 roundId) external view returns (int256);
  function getTimestamp(uint256 roundId) external view returns (uint256);

  event AnswerUpdated(int256 indexed current, uint256 indexed roundId, uint256 updatedAt);
  event NewRound(uint256 indexed roundId, address indexed startedBy, uint256 startedAt);
}

interface AggregatorV3Interface {

  function decimals() external view returns (uint8);
  function description() external view returns (string memory);
  function version() external view returns (uint256);

  // getRoundData and latestRoundData should both raise "No data present"
  // if they do not have data to report, instead of returning unset values
  // which could be misinterpreted as actual reported values.
  function getRoundData(uint80 _roundId)
    external
    view
    returns (
      uint80 roundId,
      int256 answer,
      uint256 startedAt,
      uint256 updatedAt,
      uint80 answeredInRound
    );
  function latestRoundData()
    external
    view
    returns (
      uint80 roundId,
      int256 answer,
      uint256 startedAt,
      uint256 updatedAt,
        uint80 answeredInRound
      );

}

interface AggregatorV2V3Interface is AggregatorInterface, AggregatorV3Interface {}
```

Fastlane: Partially mitigated in PR 184.

Spearbit: Verified. After a discussion with the team, the issue of unexpected behavior compared to Chainlink is not easy to mitigate completely in the current system. So, a documentation note has been added, which is especially important for protocols that use the original `roundId` system.

5.2.12 `callIndex` incremented twice

Severity: Medium Risk

Context: Atlas.sol#L237-L295, Escrow.sol#L160-L162, SafetyBits.sol#L87-L94

Description: Function `_executeSolverOperation()` increases `callIndex`. The function is also called from a loop in either `_bidFindingIteration()` or `_bidKnownIteration()`. So `callIndex` is incremented twice for every failed `SolverOp`, which doesn't seems logical.

Also see issue "`_bidFindingIteration` doesn't reset `key.callIndex`" how this is a factor in limiting the maximum solvers to 83.

```
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    for (uint256 i; i < j; i++) {
        (auctionWon, key) = _executeSolverOperation(/*...*/);
        if (auctionWon) {
            // ...
            return (auctionWon, key);
        }
    }
}
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    key = key.holdSolverLock(solverOp.solver); // increments callIndex
    // ...
    if (result.executionSuccessful()) {
        key.solverSuccessful = true;
        return (true, key);    // auctionWon = true
    }
    // ...
    ++key.callIndex; // why is this done? Is within a loop
    // ...
}
function holdSolverLock(EscrowKey memory self, address nextSolver) internal pure returns (EscrowKey
    memory) {
    // ...
    ++self.callIndex;
    // ...
}
```

Recommendation: Doublecheck the usefulness of incrementing `callIndex`. Consider removing the increment to `callIndex` from `_executeSolverOperation()`.

```
  function _executeSolverOperation(...) ... {
      // ...
-     ++key.callIndex;
      // ...
  }
```

Fastlane: Solved by a different implementation in PR 225.

Spearbit: Verified.

5.2.13 Winning solver doesn't get gas costs `_assign()`ed

Severity: Medium Risk

Context: Escrow.sol#L97-L168

Description: Function `_executeSolverOperation()` doesn't call `_releaseSolverLock()` for the winning solver so the gas costs don't get `_assign()`ed.

```
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.executionSuccessful()) {
        // ...
        key.solverSuccessful = true;
        // auctionWon = true
        return (true, key);  // no call to _releaseSolverLock
    }
    // ...
    _releaseSolverLock(solverOp, gasWaterMark, result, false, !prevalidated);
    // ...
}
```

Recommendation: Carefully check `_executeSolverOperation()` for all situations where `_releaseSolverLock()` should be called. Also check related issues:

- "Solvers don't always reimburse the bundler".
- "Function `_releaseSolverLock()` doesn't undo all the actions of `_trySolverLock()`".
- "Unreachable code in `_assign()`".

Fastlane: Solved in PR 271. `_assign()` is now called in `_settle()` at the end of the metacall, and is called in both cases where:

1) There is a winning solver and...
2) The solver is in deficit (owes money to Atlas) and not in surplus (in which case the solver gets `_credit()`ed.

Spearbit: Verified.

5.2.14 claims accounting only tracks execution costs

Severity: Medium Risk

Context: Atlas.sol#L51

Description: The claims storage variable records the ETH that will be reimbursed to the bundler for paying transaction fees. Currently, this value only tracks the difference between two calls to gasleft(), which implies that the bundler is only reimbursed for execution costs between two markers. However, there are other costs associated with being a bundler, for example, a base 21_000 gas cost for the entire transaction and an additional cost for each byte of calldata. To be more accurate, these costs could be added to the claims accounting.

It appears that this was already intended based on the following commented-out code:

```
uint256 gasMarker = gasleft(); // + 21_000 + (msg.data.length * _CALLDATA_LENGTH_PREMIUM);
```

and also based on the fact that _releaseSolverLock() may charge some solvers for their calldata costs.

Recommendation: Add the additional gas costs into the claims accounting. This can be accomplished by uncommenting the code mentioned above.

Fastlane: Fixed in PR 243.

Spearbit: Verified.

5.2.15 Incorrect SURCHARGE multiplication

Severity: Medium Risk

Context: GasAccounting.sol#L289

Description: In the _setAtlasLock() function, the following code sets claims to the maximum amount of ETH the bundler will use, including a surcharge:

```
// Set the claimed amount
uint256 rawClaims = (gasMarker + 1) * tx.gasprice;
claims = rawClaims + ((rawClaims * SURCHARGE) / 10_000_000);
```

Later on in the _settle() function, the remaining unused gas is subtracted from claims, also including the surcharge:

```
uint256 gasRemainder = (gasleft() * tx.gasprice);
gasRemainder += ((gasRemainder * SURCHARGE) / 10_000_000);
_claims -= gasRemainder;
```

Since both of these terms included the surcharge, the _claims value in _settle() ultimately represents the total ETH used by the bundler plus the surcharge amount. Therefore the following calculation is based on a combined amount, which is incorrect:

```
uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;
_claims -= netGasSurcharge;
surcharge = _surcharge + netGasSurcharge;
SafeTransferLib.safeTransferETH(bundler, _claims);
```

For example, with a 10% surcharge, this code sets netGasSurcharge to 10% of 110% of the total ETH used, which leads to the bundler only being reimbursed 99% of the ETH they spent.

Recommendation: Since _claims is already a combined value, the amount to multiply by the SURCHARGE should instead be _claims * 10_000_000 / (10_000_000 + SURCHARGE). In the netGasSurcharge calculation, this would be equivalent to making the following change:

```
- uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;
+ uint256 netGasSurcharge = (_claims * SURCHARGE) / (10_000_000 + SURCHARGE);
```

Fastlane: Solved in PR 228.

Spearbit: Verified.

5.2.16 Call to `validateUserOp()` won't work

Severity: Medium Risk

Context: AtlasVerification.sol#L532-L575

Description: `_verifyUser()` uses the ERC4337 function `validateUserOp()` to validate smart contract wallets. There are several reasons why this won't work:

- entryPoint v0.6 has a different layout for `UserOperation`.
- entryPoint v0.7 has yet another layout for `UserOperation`.
- `smartwallets` usually allow only calls from the `EntryPoint` to `validateUserOp`, see BaseAccount.sol.
- Any random smart contract that has a fallback function that returns `0` on unknown functions would satisfy this check.

Note: other erc-4337 wallets usually don't put a gas limit when calling `validateUserOp()`.

```
function _verifyUser(/*...*/) /*...*/ {
    if (userOp.from.code.length > 0) {
        // ...
        bool validSmartWallet =
            IAccount(userOp.from).validateUserOp{ gas: 30_000 }(userOp, _getProofHash(userOp), 0) == 0;
        return (isSimulation || validSmartWallet);
    }
    // ...
}
```

Recommendation: `ERC1271.isValidSignature()` seems a more logical solution. Also see OZ SignatureChecker. However be aware of implementation issues of `ERC1271.isValidSignature()`.

Fastlane: Solved by PR 250.

Spearbit: Verified.

5.2.17 V2DAppControl allows both `amount0Out` and `amount1Out` to be non-zero

Severity: Medium Risk

Context: V2DAppControl.sol#L98-L109

Description: `V2DAppControl` assumes exactly one `amount0Out` and `amount1Out` is non-zero, but this isn't enforced. If both these values are non-zero, `amount0In` and `amount1In` are calculated incorrectly:

```
uint256 amount0In =
    amount1Out == 0 ? 0 : SwapMath.getAmountIn(amount1Out, uint256(token0Balance),
↪   uint256(token1Balance));
uint256 amount1In =
    amount0Out == 0 ? 0 : SwapMath.getAmountIn(amount0Out, uint256(token1Balance),
↪   uint256(token0Balance));
```

This calculation assumes that after the swap, the pool will only transfer out exactly one token.

Recommendation: Revert if both `amount0Out` and `amount1Out` are non-zero.

Fastlane: Solved in PR 238.

Spearbit: Verified.

5.2.18 `ChainlinkAtlasWrapper` allows retransmitting old reports

Severity: Medium Risk

Context: ChainlinkAtlasWrapper.sol#L108

Description: When `transmit()` is called on the `ChainlinkAtlasWrapper`, there is nothing checking that the report and corresponding signatures haven't been used before. While the transmitters are whitelisted and trusted to an extent, this behavior means a single bad actor can exploit the system (which is otherwise secured by multiple independent parties).

Recommendation: Ensure that a `report` can only be transmitted once. This might be achieved by using a system of increasing ids, similar to the Chainlink `BASE_FEED`.

Fastlane: Solved in PR 184.

Spearbit: Verified.

5.2.19 Solvers don't always reimburse the bundler

Severity: Medium Risk

Context: Escrow.sol#L320-L387, Escrow.sol#L97-L168

Description: In the Atlas system, bundlers pay gas fees upfront and are eventually reimbursed throughout the transaction flow. This is facilitated through the `claims` storage variable (which tracks the total amount due), the `_releaseSolverLock()` function (which assigns a reimbursement amount to a specific solver), and finally the `_settle()` function (which ensures that `deposits >= withdrawals + claims`).

While this system generally assigns costs fairly, there are two situations where reimbursements are not made as expected. Both situations are the result of an early `return` that skips a call to `_releaseSolverLock()`, even though the early return may be caused by a `_PARTIAL_REFUND` error (which is expected to result in a gas reimbursement).

The first location of this issue is in the `_getBidAmount()` function, where `_releaseSolverLock()` is only reached if all validation succeeds and the `solverMetaTryCatch()` call is made. Also, note that since there are situations where gas *is* charged, there seems to be a contradiction with the following comment in the function:

```
// NOTE: To prevent a malicious bundler from aggressively collecting storage refunds,
// solvers should not be on the hook for any 'on chain bid finding' gas usage.
```

The second location of this issue is in the `_executeSolverOperation()` function, where an early return can happen if the `_handleAltOpHash()` logic fails.

Recommendation: Whenever an error leads to an early `return`, ensure that the relevant gas costs are assigned to the solver at fault. This can be achieved by adding a call to `_releaseSolverLock()` before each early return, or potentially by reworking the way that gas costs are allocated.

Fastlane: Fixed in PR 271. Also added a follow-up change to assign the bid-finding gas costs to the bundler in PR 371.

Spearbit: Verified.

5.2.20 Deadline check skipped in simulation mode

Severity: Medium Risk

Context: AtlasVerification.sol#L132-L139

Description: Deadline check for `userOp` and `dAppOp` is skipped in simulation mode. This shouldn't be the case as a successful simulation will lead to an onchain transaction which will then revert wasting gas for the bundler.

Recommendation: Check against deadline for simulation mode:

```
- if (userOp.deadline != 0 && !isSimulation) {
+ if (userOp.deadline != 0) {
      return (userOpHash, ValidCallsResult.UserDeadlineReached);
  }
  // ...
- if (dAppOp.deadline != 0 && !isSimulation) {
+ if (dAppOp.deadline != 0) {
      return (userOpHash, ValidCallsResult.DAppDeadlineReached);
  }
```

Fastlane: Solved in PR 178.

Spearbit: Verified.

5.2.21 `amount` is downcasted to `uint112` without overflow protection

Severity: Medium Risk

Context: AtlETH.sol

Description: `amount` is downcasted from `uint256` to `uint112` at various places in `AtlEth.sol` as highlighted above. `transfer()` and `transferFrom()` could do an `emit` with a very large amount if passing an amount such as `type(uint112).max + 1`. This will confuse chain indexers.

```
function _deduct(address account, uint256 amount) internal {
    uint112 amt = uint112(amount);
    // ...
    revert InsufficientBalanceForDeduction(/*...*/ , amount); // possibly large amount
}
function _burn(address from, uint256 amount) internal {
    _deduct(from, amount);
    totalSupply -= amount;  // will fail with large amount
    // ...
}
function transfer(address to, uint256 amount) public returns (bool) {
    _deduct(msg.sender, amount);
    _balanceOf[to].balance += uint112(amount);
    emit Transfer(msg.sender, to, amount); // could do emit with large amount
    // ...
}
function transferFrom(address from, address to, uint256 amount) public returns (bool) {
    uint256 allowed = allowance[from][msg.sender]; // Saves gas for limited approvals.
    if (allowed != type(uint256).max)
        allowance[from][msg.sender] = allowed - amount;  // could fail
    _deduct(from, amount);
    _balanceOf[to].balance += uint112(amount);
    emit Transfer(from, to, amount); // could do emit with large amount
    return true;
}
```

In `_unbond()` an artificial amount is emitted when passing an amount such as `type(uint112).max + 1`.

```
function unbond(uint256 amount) external {
    _unbond(msg.sender, amount);
}
function _unbond(address owner, uint256 amount) internal {
    uint112 amt = uint112(amount); // can be truncated
    // ...
    emit Unbond(owner, amount, block.number + ESCROW_DURATION + 1);
}
```

Function `_mint()` also does the downcast. But this won't happen in practice because it is only called via `deposit()` and `depositAndBond()` which are bounded by `msg.value`.

```
function _mint(address to, uint256 amount) internal {
    totalSupply += amount;
    _balanceOf[to].balance += uint112(amount);
    emit Transfer(address(0), to, amount);
}
```

Recommendation: Use `SafeCast.toUint112` to safely downcast. Now if `amount` exceeds `uint112`'s max value, it reverts.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.2.22 Solver bundler doesn't enforce exactly one `solverOps`

Severity: Medium Risk

Context: AtlasVerification.sol#L231-L253

Description: Function `_verifyAuctioneer()` use `solverOps[0]`. This will `revert` when there are no `solverOps`. *Note: this can happen when `allowsZeroSolvers()==true`.*

As we understood from the `Fastlane` project, when a `solver` is also a `bundler` there should be exactly one `solverOps`. This isn't enforced in the code.

```
function _verifyAuctioneer(/*...*/) /*...*/ {
    // ...
    if (dConfig.callConfig.allowsSolverAuctioneer() && dAppOp.from == solverOps[0].from) return (true,
↪   true);
    // ...
}
```

Recommendation: Enforce `solverOps.length == 1` when the `solver` is the `bundler`. The best location for this is probably `verifySolverOp()`.

Fastlane: Solved in PR 172. The specific rule we'll enforce here is: if the solver is the auctioneer, then there must be exactly 1 solver. However, a call config may be set to `allowsSolverAuctioneer == true`, and the `auctioneer` could still be another party e.g. the user if `allowsUserAuctioneer == true` as well. In this case where the `auctioneer` is not the `solver`, we should still allow multiple `solverOps`.

The extra checks are done in `_verifyAuctioneer()` instead of `verifySolverOp()` as a deeper refactor is needed to get around Stack Too Deep to pass the additional params needed to handle this in `verifySolverOp()`.

Spearbit: Verified.

5.2.23 External calls may use more gas than `gasLimit`

Severity: Medium Risk

Context: ExecutionEnvironment.sol#L213

Description: In the following case, `.call` copies the entire return data to memory even if it isn't used:

```
(success,) = solverOp.solver.call{ gas: gasLimit, value: solverOp.value }(solverCallData);
```

Since this leads to memory expansion costs, this call may use significantly more gas than just the `gasLimit` value. The `solverGasLimit` is used in `_validateSolverOperation()` to ensure a solver has sufficient funds, so it's likely unexpected for this call to use extra gas. Although data isn't supposed to be returned in this call, it may be done intentionally by an adversarial solver to grief the system.

Recommendation: `ExcessivelySafeCall` is a library to restrict the return data to a particular size. This library was created to prevent "returnbombing" where the callee can force the caller to copy large amount of data wasting caller's gas and potentially halt execution.

`ExcessivelySafeCall` does some extra operations which are needed only if some return data is needed. Alternative is to use or copy `call()` fn from `SafeCall.sol` which doesn't do any operation related to copying return data saving gas.

Fastlane: Solved in PR 272 using OP's `SafeCall` lib as we don't need return data from solver calls.

Spearbit: Verified.

5.2.24 `reconcile()` can be called by anyone

Severity: Medium Risk

Context: GasAccounting.sol#L73-L119, GasAccounting.sol#L208-L215, GasAccounting.sol#L295, Atlas.sol#L91

Description: Anyone can call `reconcile()` because the the checks on `lock` and `currentSolver` are done with user supplied parameters. These checks can even pass when `lock == UNLOCKED` or `_solverLock == _UNLOCKED_UINT`.

Function `reconcile()` can set the flags `_solverCalledBack` and `_solverFulfilled`. Luckily `_trySolverLock()` resets these flags. Although `_trySolverLock()` isn't always done, as show in the issue "`_releaseSolverLock()` can be run without `_trySolverLock()`". Currently that doesn't create an issue.

Function `reconcile()` can do several unwanted actions:

- `reconcile()` creates `deposits` out of thin air
- Flag `_solverFulfilled` is unreliable

After `validateBalances` then `_solverFulfilled` is not used anymore. After `_settle()` then `deposits` is not used anymore.

Places where `reconcile()` can be called:

- Before the call to `metacall()` → not an issue.
- In PreOps hook → before `validateBalances` and `_settle()` so is an issue.
- In UserOp hook → before `validateBalances` and `_settle()` so is an issue.
- In Solver / PreSolver → not an issue because then it is supposed to happen.
- In `AllocateValue` → before `_settle()` so is an issue.
- In `PostOps` → before `_settle()` so is an issue.
- Via `safeTransferETH()` of `_settle()` → after the relevant logic of `_settle()` so is no issue.
- Via `safeTransferETH()` of `metacall()` → `deposits` not used → no issue.

```
function reconcile(address environment, address solverFrom,...) ... {
    // ...
    if (lock != environment) revert InvalidExecutionEnvironment(lock); // environment is user supplied
    (address currentSolver, bool calledBack, bool fulfilled) = solverLockData();
    if (solverFrom != currentSolver) revert InvalidSolverFrom(currentSolver); // solverFrom is user
↪   supplied
    // ...
    _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack;
    // ...
    _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack | _solverFulfilled;
}
function _trySolverLock(SolverOperation calldata solverOp) internal returns (bool valid) {
    if (_borrow(solverOp.value)) {
        _solverLock = uint256(uint160(solverOp.from)); // resets  flags `_solverCalledBack` and
↪   `_solverFulfilled`
        // ...
    } else {
        // ...
    }
}
```

Recommendation: Access to `reconcile()` should be restricted to specific callers and phases. Also see issues:

- Future authorization might fail because solver contract isn't `solverOp.from`
- Locking mechanism is complicated

Fastlane: Resolved in PR 271 and PR 287. Only the solver's `solverOp.solver` address can call `reconcile()` during their `SolverOperation` phase.

Spearbit: Verified.

5.2.25 A `solver` with insufficient funds can block further processing

Severity: Medium Risk

Context: GasAccounting.sol#L254-L298, GasAccounting.sol#L225-L246

Description: Function `_settle()` reverts if the `Solver` can't pay for the costs. When function `_settle()` reverts then `metacall()` also reverts. The costs could be: `gas` usage or any `Borrow()`s after `validateBalances()`.

This way a `solver` with insufficient funds can block further processing of the other `solvers`. However a user would expect that when a `solver` fails, then next `solver` in the list would be used.

In comparison: When a `solver` doesn't win, and via `_releaseSolverLock()`, the gas `_assign()`ment fails, then that error is ignored.

Note: `Borrow()`s after `validateBalances()` are questionable, see issue "`Borrow()`s after `validateBalances()`".

```
function _settle(/*...*/) /*...*/ {
    // ...
    if (_assign(winningSolver, amountOwed, true, false)) {
        revert InsufficientTotalBalance((_claims + _withdrawals) - deposits);
    }
    // ...
}

function _releaseSolverLock(/*...*/) /*...*/ {
    // ...
    _assign(solverOp.from, gasUsed, false, bidFind); // failure to assign is ignored
}
```

Recommendation: Consider disallowing `Borrow()`s after `validateBalances()`, see issue "`Borrow()`s after `validateBalances()`".

Consider wrapping all `solver` related actions in a `try/catch`. For example by doing the following steps in `solverMetaTryCatch()` too:

- `_allocateValue()`
- `_executePostOpsCall()`
- `_settle()`

Fastlane: Solved in PR 227:

1) `reconcile()` *must* be called, and only during the `SolverOperation` phase, and...
2) `borrow()` is blocked after `reconcile()` has been called, even if it is still during the `SolverOperation` phase.

Spearbit: Verified.

5.3 Low Risk

5.3.1 Remove `Test` inheritance

Severity: Low Risk

Context: SolverBase.sol#L18

Description: `SolverBase` inherits from `Test` contract:

```
contract SolverBase is Test {
```

This increases the contract size and may expose any unsafe functionality.

Recommendation: Remove `Test` inheritance.

Fastlane: Solved in PR 187 and PR 342.

Spearbit: Verified.

5.3.2 `disableDApp()` doesn't clean up `dAppSignatories[]`

Severity: Low Risk

Context: DAppIntegration.sol#L122-L131, DAppIntegration.sol#L150-L160

Description: `disableDApp()` doesn't clean up `dAppSignatories[]` like `_removeSignatory()` does. This could be a problem if the dapp would be enabled again, then `dAppSignatories[]` would contain the same `govAddress` address twice. Also `getDAppSignatories()` doesn't give an accurate view.

Function `disableDApp()` doesn't check the `signatoryKey` was enabled, like `changeDAppGovernance()` does. This could result in redundant `emit`s.

```
function disableDApp(address dAppControl) external {
    // ...
    signatories[signatoryKey] = false;
    //... // no clean up of dAppSignatories[]
}
function _removeSignatory(address controller, address signatory) internal {
    // ...
    delete signatories[signatoryKey];
    for (uint256 i = 0; i < dAppSignatories[controller].length; i++) {
        if (dAppSignatories[controller][i] == signatory) {
            dAppSignatories[controller][i] =
↪   dAppSignatories[controller][dAppSignatories[controller].length - 1];
            dAppSignatories[controller].pop();
            break;
        }
    }
}
```

Recommendation: Consider calling `_removeSignatory()` from `disableDApp()`. Also see the issue "`_removeSignatory()` can silently fail".

Fastlane: Solved by PR 189.

Spearbit: Verified.

5.3.3 Use of `storage` variables versus `delegatecall`

Severity: Low Risk

Context: DAppControl.sol#L20-L28, SwapIntent.sol#L66, ChainlinkDAppControl.sol#L37-L81

Description: The usage of `storage` with `DAppControl` is not trivial:

- If `DAppControl` based contracts use a `storage` variable it wil be stored in the `ExecutionEnvironment` and it can be changed by a user contract if called via `delegatecall`.
  - Contract `ChainlinkDAppControl` uses storage variable `verificationVars` (but luckily `delegateUser: false`).
  - Contract `SwapIntentController` allows `delegatecall` via `delegateUser: true` (but luckily no storage variables).
- DAppControl has two storage variables: `governance` and `pendingGovernance`, which means all functions that access these should not be `delegatecall`ed.
  - The functions `getDAppSignatory()`, `transferGovernance()` and `acceptGovernance()` don't have the modifier `mustBeCalled` so could accidentally be called via `delegatecall`.

```
abstract contract DAppControl is DAppControlTemplate, ExecutionBase {
    // ...
    address public governance;
    address public pendingGovernance;
    // ...
}
contract SwapIntentController is DAppControl {
     constructor(address _atlas) DAppControl(_atlas, msg.sender, CallConfig({
                // ...
                delegateUser: true,
                // ...
            })
        )
    // ...
}
contract ChainlinkDAppControl is DAppControl {
    // ...
    mapping(address baseChainlinkFeed => VerificationVars) internal verificationVars; // storage
    constructor(address _atlas) DAppControl(_atlas, msg.sender, CallConfig({
                // ...
                delegateUser: false,
                // ...
            })
        )
    // ...
}
```

Recommendation: Document the use of `storage` variables versus `delegatecall`, also taking into account usage in an indirect way, for example via OZ ReentrancyGuard. Consider adding the modifier `mustBeCalled` to `getDAppSignatory()`, `transferGovernance()` and `acceptGovernance()`.

Fastlane: Solved in PR 263.

Spearbit: Verified.

5.3.4 `V2DAppControl _preOpsCall()` doesn't check destination for call

Severity: Low Risk

Context: V2DAppControl.sol#L86-L113

Description: `_preOpsCall()` calls a function from `userOp.dapp` but doesn't check if it is a valid uniswap V2 compatible pair.

```
function _preOpsCall(UserOperation calldata userOp) internal override returns (bytes memory) {
    // ...
    (uint112 token0Balance, uint112 token1Balance,) = IUniswapV2Pair(userOp.dapp).getReserves();
    // ...
    _transferUserERC20(
        amount0Out > amount1Out ? IUniswapV2Pair(userOp.dapp).token1() :
↪   IUniswapV2Pair(userOp.dapp).token0(),
        userOp.dapp,
        amount0In > amount1In ? amount0In : amount1In
    );
    // ...
}
```

Recommendation: Consider checking `userOp.dapp` is a valid uniswap V2 compatible pair, for example via the factory.

Fastlane: Solved in PR 204.

Spearbit: Verified.

5.3.5 No validity check on `chainlinkWrapper`

Severity: Low Risk

Context: ChainlinkDAppControl.sol#L87-L91

Description: In `_allocateValueCall()` there is no check done that `chainlinkWrapper` is valid.

```
function _allocateValueCall(address bidToken, uint256 bidAmount, bytes calldata data) internal virtual
↪   override {
    address chainlinkWrapper = abi.decode(data, (address));
    (bool success,) = chainlinkWrapper.call{ value: bidAmount }("");
    if (!success) revert FailedToAllocateOEV();
}
```

Recommendation: Consider checking `chainlinkWrapper` is one of the valid wrappers.

Fastlane: Solved in PR 184.

Spearbit: Verified.

5.3.6 `CallValueTooHigh` error calculation is incorrect

Severity: Low Risk

Context: Escrow.sol#L275-L280

Description: In the `_validateSolverOperation()` function, the following check verifies that `solverOp.value` is larger than `address(this).balance` minus a gas amount:

```
// Verify that we can lend the solver their tx value
if (
    solverOp.value
        > address(this).balance - (gasLimit * tx.gasprice > address(this).balance ? 0 : gasLimit *
↪   tx.gasprice)
) {
    return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
}
```

In this calculation, the `gasLimit * tx.gasprice > address(this).balance` check appears to prevent a subtraction underflow if `gasLimit * tx.gasprice` is larger than `address(this).balance`. However, in the case where the underflow would happen, the subtracted amount is 0 (which results in `solverOp.value > address(this).balance`) when it was likely intended to be `address(this).balance` (which results in `solverOp.value > 0`).

Moreover, it's not clear if this subtraction is completely necessary for this check. Since the transaction gas costs will not decrease `address(this).balance`, and since all borrowed ETH and gas refunds are guaranteed to be paid at the end of an Atlas transaction, it may be possible to simplify the check.

Recommendation: Change the subtraction amount to be `address(this).balance` if `gasLimit * tx.gasprice > address(this).balance`:

```
if (
    solverOp.value
        > address(this).balance - (gasLimit * tx.gasprice > address(this).balance ?
↪   address(this).balance : gasLimit * tx.gasprice)
) {
    return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
}
```

Also, consider if the check can be simplified as follows:

```
if (solverOp.value > address(this).balance) {
    return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
}
```

Fastlane: Solved in PR 223.

Spearbit: Verified.

5.3.7 Signatures may be reused between the `ChainlinkAtlasWrapper` and `BASE_FEED`

Severity: Low Risk

Context: ChainlinkAtlasWrapper.sol#L77-L82

Description: The `ChainlinkAtlasWrapper` is intended to be a wrapper of the Chainlink `BASE_FEED` contract, with the two contracts potentially sharing the same signers and transmitters. Since both contracts have the same arguments and verification logic in the `transmit()` function, it seems that the `report` and corresponding signatures for one contract can also be used in the other contract.

This may not be intended, and may add a trust assumption that the transmitter relays information to the correct contract that the signers are expecting.

Recommendation: Consider adding specific logic in the `ChainlinkAtlasWrapper` that ensures the submitted `report` would not also be valid in the `BASE_FEED`. For example, this can be accomplished by enforcing that `rawObservers` from the following code is equal to `bytes32(0)`:

```
(r.rawReportContext, rawObservers, r.observations) = abi.decode(
  _report, (bytes32, bytes32, int192[])
);
```

This would work because the `BASE_FEED` would interpret this value as duplicate zero indices (which leads to a revert if there's more than one signer), while the `rawObservers` is otherwise unused in the `ChainlinkAtlasWrapper`. Therefore, it would be impossible to have one `report` be valid in both contracts if it was required that `rawObservers == bytes32(0)` in the `ChainlinkAtlasWrapper`.

Fastlane: Acknowledged. The issue with using a special rawObservers value (or any special change that would break the normal verification of the transmission in the base Chainlink contract) is that this changes the report data, and the hash of the report is what the Chainlink nodes sign when submitting a new price observation. Each signer would need to re-sign a price specifically intended for the Atlas OEV system. So while this would be better security, it would be a more onerous burden on the Chainlink system to enable OEV capture through Atlas.

As mentioned in this issue, both the Chainlink and Atlas transmit() functions are permissioned, so there are some trust assumptions to fall back on.

Will acknowledge and leave the related code as it is for now, but in the case that Chainlink nodes are willing to sign special Atlas price observations as well as their usual Chainlink ones, we will implement this safeguard.

Spearbit: Acknowledged.

5.3.8 claims accounting does not track all execution costs

Severity: Low Risk

Context: GasAccounting.sol#L272-L297

Description: To facilitate bundler gas reimbursements, the claims storage variable tracks the gas costs between two different gasleft() checkpoints. Since the gas costs incurred after the second checkpoint are not tracked, there is some amount of gas that the bundler is not reimbursed. Currently, this amounts to all of the following code within _settle():

```
gasRemainder += ((gasRemainder * SURCHARGE) / 10_000_000);
_claims -= gasRemainder;

if (_deposits < _claims + _withdrawals) {
    // CASE: in deficit, subtract from bonded balance
    uint256 amountOwed = _claims + _withdrawals - _deposits;
    if (_assign(winningSolver, amountOwed, true, false)) {
        revert InsufficientTotalBalance((_claims + _withdrawals) - deposits);
    }
} else {
    // CASE: in surplus, add to bonded balance
    // TODO: make sure this works w/ the surcharge 10%
    uint256 amountCredited = _deposits - _claims - _withdrawals;
    _credit(winningSolver, amountCredited);
}

uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;

_claims -= netGasSurcharge;

surcharge = _surcharge + netGasSurcharge;

SafeTransferLib.safeTransferETH(bundler, _claims);

return (_claims, netGasSurcharge);
```

and also includes the cost of later emitting the MetacallResult() event and calling _releaseAtlasLock(). While these are not necessarily large costs, making the gas accounting more fair for the bundler may be possible.

Recommendation: To account for these untracked costs, consider adding a fixed amount to the claims variable at the start of an Atlas transaction. This amount would preferably be an upper bound on the gas used by the final execution, except for the safeTransferETH() call which may use an unpredictable amount of gas (but is in the bundler's control). Based on the current test cases, an offset of 100_000 gas would be appropriate, since all test cases use less than this amount after the second gasleft(). Adding this fixed amount earlier is preferred, so it's part of the validateBalances() check in the ExecutionEnvironment.

Fastlane: Solved in PR 236.

Spearbit: Verified.

5.3.9 Similar functions `pack()` and `_firstSet()`/`_firstSetSpecial()` use different patterns

Severity: Low Risk

Context: SafetyBits.sol#L52-L67, ExecutionBase.sol#L49-L100

Description: The similar functions `pack()` and `_firstSet()`/`_firstSetSpecial()` use different patterns. `pack()` uses a typecast to `bytes32()` while the other functions don't.

Note: the typecast to `bytes32()` truncates the data if it is larger than 32 bytes, which isn't the case here.

```
function pack(EscrowKey memory self) internal pure returns (bytes32 packedKey) {
    packedKey = bytes32(  //bytes32 not present in other functions and truncates data
        abi.encodePacked(
            self.addressPointer,
                // ...
        )
    );
}
function _firstSet() internal pure returns (bytes memory data) {
    data = abi.encodePacked(
        _addressPointer(),
        // ...
    );
}
function _firstSetSpecial(ExecutionPhase phase) internal pure returns (bytes memory data) {
    // ...
    data = abi.encodePacked(
        _addressPointer(),
        // ...
    );
}
```

Recommendation: Consider using the same pattern for all comparable functions.

Fastlane: Solved in PR169 and PR 227 by removing `_firstSetSpecial()`.

Spearbit: Verified.

5.3.10 `USER_TYPE_HASH` and `SOLVER_TYPE_HASH` define `data` as `bytes32`

Severity: Low Risk

Context: UserCallTypes.sol#L4-L6, SolverCallTypes.sol#L4-L6, AtlasVerification.sol#L580-L597, AtlasVerification.sol#L273-L291

Description: The `USER_TYPE_HASH` and `SOLVER_TYPE_HASH` define `data` as `bytes32`, while in reality it is `bytes`. `_getProofHash()` and `_getSolverHash()` do already hash the data.

```
bytes32 constant USER_TYPE_HASH = keccak256("UserOperation(... ,bytes32 data)");
bytes32 constant SOLVER_TYPE_HASH = keccak256("SolverOperation(... ,bytes32 data)");

function _getProofHash(UserOperation memory userOp) internal pure returns (bytes32 proofHash) {
    proofHash = keccak256(
        abi.encode(
            // ...
            keccak256(userOp.data)
        )
    );
}
function _getSolverHash(SolverOperation calldata solverOp) internal pure returns (bytes32 solverHash) {
    return keccak256(
        abi.encode(
            // ...
            keccak256(solverOp.data)
        )
    );
}
```

Recommendation: Consider changing the type hashes to:

```
- bytes32 constant USER_TYPE_HASH = keccak256("UserOperation(... ,bytes32 data)");
+ bytes32 constant USER_TYPE_HASH = keccak256("UserOperation(... ,bytes data)");
- bytes32 constant SOLVER_TYPE_HASH = keccak256("SolverOperation(... ,bytes32 data)");
+ bytes32 constant SOLVER_TYPE_HASH = keccak256("SolverOperation(... ,bytes data)");
```

Fastlane: Solved in PR 174 and PR 297.

Spearbit: Verified.

5.3.11 SOLVER_TYPE_HASH contains different field than SolverOperation

Severity: Low Risk

Context: SolverCallTypes.sol#L4-L23

Description: The SOLVER_TYPE_HASH contains dapp, wheras the struct SolverOperation has solver at the same location.

```
bytes32 constant SOLVER_TYPE_HASH = keccak256(
    "SolverOperation(address from,
        ...
        uint256 deadline,
        address dapp, // should probably be solver
        address control,
        ...
    )"
);
struct SolverOperation {
    // ...
    uint256 deadline;
    address solver;
    address control;
   // ...
}
```

Recommendation: Consider replacing dapp with solver.

```
  bytes32 constant SOLVER_TYPE_HASH = keccak256(
      "SolverOperation(address from,
          ...
          uint256 deadline,
-         address dapp,
+         address solver,
          address control,
          ...
      )"
  );
```

Fastlane: Solved in PR 175.

Spearbit: Verified.

5.3.12 Statistics for auctionWins and auctionFails are inaccurate

Severity: Low Risk

Context: GasAccounting.sol#L143-L202

Description: The function _assign() keeps statistics for auctionWins and auctionFails. The inverse function _credit() doesn't keep statistics.

```
function _assign(/*...*/) /*...*/ {
    // ...
    if (solverWon) {
        aData.auctionWins++;
    } else if (!bidFind) {
        aData.auctionFails++;
    }
    // ...
}
function _credit(address owner, uint256 amount) internal {
    // ... // no statistics
}
```

Recommendation: Also include statistics updates for _credit(). Perhaps it's easier to move the statistics updates to a separate function.

Also see issue "totalGasUsed is inaccurate".

Fastlane: Solved in PR 330.

Spearbit: Verified.

5.3.13 totalGasUsed is inaccurate

Severity: Low Risk

Context: GasAccounting.sol#L143-L185

Description: In function _assign(), amount can be changed if the solver has insufficient funds. The totalGasUsed uses the corrected version, because it is mean to analytics, the original value is probably better.

The inverse function _credit() doesn't keep statistics.

Furthermore, _assign() is called for two purposes. One to assign gas costs and one to assign missing ETH. Only the first one seems relevant for analytics.

```
function _assign(address owner, uint256 amount, bool solverWon, bool bidFind) internal returns (bool
↪   isDeficit) {
    // ...
    if (bData.unbonding + aData.bonded < amt) {
        // ...
        amount = uint256(bData.unbonding + aData.bonded); // contribute less to deposits ledger
      // ...
    } ...
    // Reputation Analytics: Track total gas used, solver wins, and failures
    aData.totalGasUsed += uint64(amount / GAS_USED_DECIMALS_TO_DROP);
    // ...
}
function _credit(address owner, uint256 amount) internal {
    // ... // no statistics for totalGasUsed
}
```

Recommendation: Only increase `totalGasUsed` with the gas costs, for example use `_claims` after `gasRemainder` is deducted. And use `amt` in the following way:

```
- aData.totalGasUsed += uint64(amount / GAS_USED_DECIMALS_TO_DROP);
+ aData.totalGasUsed += uint64(amt / GAS_USED_DECIMALS_TO_DROP);
```

Also include statistics updates for `_credit()`. Perhaps its easier to move the statistics updates to a separate function.

Also see issue "Statistics for `auctionWins` and `auctionFails` not accurate".

Fastlane: Solved in PR 330.

Spearbit: Verified.

5.3.14 `userWrapper()` does not always need `forward()` data

Severity: Low Risk

Context: ExecutionEnvironment.sol#L103-L110

Description: At the end of the `userWrapper()` function, a `call` or `delegatecall` is made to the `userOp.dapp` address:

```
if (config.needsDelegateUser()) {
    (success, returnData) = userOp.dapp.delegatecall(forward(userOp.data));
    require(success, "ERR-EC02 DelegateRevert");
} else {
    // regular user call - executed at regular destination and not performed locally
    (success, returnData) = userOp.dapp.call{ value: userOp.value }(forward(userOp.data));
    require(success, "ERR-EC04a CallRevert");
}
```

Notice that this call will use the `forward()` helper function. This function appends extra data (e.g. address pointers, call depth, etc) so that Atlas-specific contracts can inspect the state of the call.

However, the `userOp.dapp` address may not be an Atlas-specific contract. For example, with the `V2DAppControl`, the `userOp.dapp` address would be a UniswapV2 pool. As a result, the extra calldata will not always be used or expected. In rare scenarios, this might cause reverts in protocols that have unique calldata expectations.

Recommendation: Consider removing the `forward()` functionality whenever `userOp.dapp != userOp.control`.

Fastlane: Solved in PR 170 and due to refactoring.

Spearbit: Verified.

5.3.15 Balance diff considerations

Severity: Low Risk

Context: ExecutionEnvironment.sol#L142-L317

Description: In the `solverMetaTryCatch()` function, the `ExecutionEnvironment` tracks the difference in its bid token balance before and after the solver receives control flow. The assumption is that an increase in token balance would be due to a direct transfer from the solver.

However, there are niche situations where this assumption might not hold. For example, if the `ExecutionEnvironment` becomes eligible for an airdrop, and if the airdrop transfer can be triggered by an arbitrary address (this is how the Uniswap `MerkleDistributor` works), then solvers might trigger the airdrop to subsidize their bid. This would be unexpected, as the airdrop already belongs to the `ExecutionEnvironment`, but is not explicitly part of its balance.

If the `ExecutionEnvironment` was used more generally as a smart contract wallet, there may be other ways that a balance diff becomes problematic. For example, if the `ExecutionEnvironment` has permitted a non-Atlas protocol to exchange one of its tokens, fulfilling that order could increase the bid token balance, and wouldn't be related to the solver's actions. This is similar to a bug that appeared in UniswapX.

Recommendation: Consider if any of the above situations might be problematic enough to warrant a change. If the situations seem niche and unlikely, consider documenting this risk. If the problem seems more severe, consider addressing this behavior by using the ERC20 `transferFrom()` function to "pull" each solver's bid directly from them.

Fastlane: Solved in PR 274 by documenting this.

Spearbit: Verified.

5.3.16 `_credit()` deviates from logic in `_assign()`

Severity: Low Risk

Context: GasAccounting.sol#L143-L202, GasAccounting.sol#L254-L298

Description: Function `_assign()` updates `deposits` but the mirror function `_credit()` doesn't update `withdrawals`. As can be called from multiple locations it is important there. `_credit()` can only be called from via `_settle()`. However, after this call there is an external call via `safeTransferETH()` so it is potentially risky to not update `withdrawals`. See issue "Call to `safeTransferETH` can do unwanted actions".

As far as we can see, no harm can be done.

```
function _settle(/*...*/) /*...*/ {
    // ...
    if (_deposits < _claims + _withdrawals) {
        // ...
        if (_assign(winningSolver, amountOwed, true, false)) {
            revert InsufficientTotalBalance((_claims + _withdrawals) - deposits); // uses updated deposits
        }
    } else {
        // ...
        _credit(winningSolver, amountCredited);
    }
    // ...
    SafeTransferLib.safeTransferETH(bundler, _claims);
    // ...
}
function _assign(address owner, uint256 amount, bool solverWon, bool bidFind) internal returns (bool isDeficit) {
    // ...
    bondedTotalSupply -= amount;
    deposits += amount;
}
function _credit(address owner, uint256 amount) internal {
    // ...
    bondedTotalSupply += amount;
    // ...  // no change in withdrawals
}
```

Recommendation: Double check there are not side effects of not increasing `withdrawals`. Consider adding a comment in function `_credit()` and/or `_assign()`.

Fastlane: Solved in PR 246.

Spearbit: Verified.

5.3.17 Special cases for `deadline == 0`

Severity: Low Risk

Context: AtlasVerification.sol#L130-L142, Escrow.sol#L256, Escrow.sol#L394-L411

Description: It seems `userOp.deadline==0` and `dAppOp.deadline==0` indicate there is no deadline. However there is no special case for `solverOp.deadline == 0`.

`_handleAltOpHash()` enforces the deadlines of `solverOp` and `userOp.deadline` to be the same. So could be an issue if `solverOp.deadline == 0` isn't supported.

```
function _validCalls(
    // ...
    if (block.number > userOp.deadline) {
        if (userOp.deadline != 0 && !isSimulation) {
            return (userOpHash, ValidCallsResult.UserDeadlineReached);
        }
    }
    if (block.number > dAppOp.deadline) {
        if (dAppOp.deadline != 0 && !isSimulation) {
            return (userOpHash, ValidCallsResult.DAppDeadlineReached);
        }
    }
    // ...
}
function _validateSolverOperation(
  // ...
    if (block.number > solverOp.deadline) { // no exception for 0
        return (/*...*/);

    }
    // ...
}
function _handleAltOpHash(/*...*/) /*...*/ {
    // ...
    if (solverOp.deadline != userOp.deadline || solverOp.control != userOp.control) {
        return false;
    }
// ...
}
```

Recommendation: Consider also supporting `solverOp.deadline == 0`. Update the check in `_handleAltOpHash()` to support one of deadlines to be 0.

Fastlane: Solved in PR 179.

Spearbit: Verified.

5.3.18 `_handleAltOpHash()` executed even in error situations

Severity: Low Risk

Context: Escrow.sol#L320-L387, Escrow.sol#L97-L168

Description: If `_validateSolverOperation()` fails then `_handleAltOpHash()` is still executed. There are two main reasons for `_validateSolverOperation()` to fail:

- `block.number` related. Error with `block.number` don't seem to be good reason to still do `_handleAltOpHash()` because this prevents executing the `solverOp` on a later moment in time.

  Note: also see a suggestion to move the block.number related checks in issue "Difference between `Sorter` and `Atlas` functions".

- Gas related. This might be a good reason.

In `_executeSolverOperation()`, when `_handleAltOpHash()` fails then the `result` of `_validateSolverOperation()` is returned an no additional error bit for the failing of `_handleAltOpHash()` is set.

```
function _getBidAmount(/*...*/) /*...*/ {
    // ...
    (result, gasLimit) = _validateSolverOperation(dConfig, solverOp, gasWaterMark, result);
    if (dConfig.callConfig.allowsTrustedOpHash()) {
        if (!_handleAltOpHash(userOp, solverOp)) {
            return (0);
        }
    }
    // ...
}
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    (result, gasLimit) = _validateSolverOperation(dConfig, solverOp, gasWaterMark, result);
    if (dConfig.callConfig.allowsTrustedOpHash()) {
        if (!prevalidated && !_handleAltOpHash(userOp, solverOp)) { // doesn't add its own error bit
            key.solverOutcome = uint24(result); // result is off the previous action
            return (false, key);
        }
    }
}
```

Recommendation: Double check the reasons for doing `_handleAltOpHash()` after an error. If there is no good reason, return with the error directly after `_validateSolverOperation()`.

In `_executeSolverOperation()`, when `_handleAltOpHash()` fails: add a specific error code.

Fastlane: Under normal operation, a given solverOp can only ever be used for a specific userOp. This means that unless `allowsReuseUserOps` is set, then a solverOp has replay protection by virtue of the userOp it references.

When `allowsTrustedOpHash` is set, this replay protection no longer works since the solverOp can now reference multiple userOps, opening up a replay attack vector if new userOps can be signed with the same trusted ophash.

The intent of this change, is that if `allowsTrustedOpHash` is set, then a solverOp should only ever be able to be included in a single atlas bundle, regardless of whether or not it executes successfully.

Given this, it seems that both the gas failure case and the deadline failure case should trigger this behavior.

Spearbit: Acknowledged.

5.3.19 Unreachable code in `_assign()`

Severity: Low Risk

Context: GasAccounting.sol#L143-L185, GasAccounting.sol#L225-L298

Description: Function `_assign()` can be called from `_releaseSolverLock()` and `_settle()`. In both cases `amount` will not be 0. See pieces of code below. This means the code `if (amount == 0) { ... }` will never be executed.

This is fortunate though because:

- `_bidFindingIteration()` calls `_getBidAmount()`, which calls `_releaseSolverLock()` which calls `_assign()`.
- `_assign()` would maybe set `lastAccessedBlock == block.number`.
- Then `_bidFindingIteration()` continues and calls `_executeSolverOperation()` which calls `_validateSolverOperation()`.
- `_validateSolverOperation()` checks `lastAccessedBlock == block.number`, which would be `true` now and result in an error.

```
function _assign(address owner, uint256 amount, bool solverWon, bool bidFind) internal returns (bool isDeficit) {
    if (amount == 0) {
        accessData[owner].lastAccessedBlock = uint32(block.number); // still save on bidFind
    } else {
        // ...
        if (!bidFind) {
            aData.lastAccessedBlock = uint32(block.number);
        }
    }
}
```

```
uint256 gasWaterMark = gasleft();
function _releaseSolverLock(/*...*/, uint256 gasWaterMark, /*...*/) /*...*/ {
    // ...
    uint256 gasUsed = (gasWaterMark - gasleft() + 5000) * tx.gasprice;
    // other action to increase gasUsed
    _assign(/*...*/, gasUsed, /*...*/);  // gasUsed at least 5000 * tx.gasprice
    // ...
}
```

```
function _settle(/*...*/) /*...*/ {
    // ...
    if (_deposits < _claims + _withdrawals) {
        uint256 amountOwed = _claims + _withdrawals - _deposits;
        if (_assign(/*...*/ , amountOwed, /*...*/)) { /*...*/ } // amountOwed > 0 otherwise doesn't end up here
    }
    // ...
}
```

Recommendation: Doublecheck the potential goal for the `amount == 0` check. Remove the code if not relevant.

Fastlane: Solved in PR 242.

Spearbit: Verified.

5.3.20 `_removeSignatory()` can silently fail

Severity: Low Risk

Context: DAppIntegration.sol#L150-L160

Description: The `_removeSignatory()` function has the following implementation:

```
function _removeSignatory(address controller, address signatory) internal {
    bytes32 signatoryKey = keccak256(abi.encodePacked(controller, signatory));
    delete signatories[signatoryKey];
    for (uint256 i = 0; i < dAppSignatories[controller].length; i++) {
        if (dAppSignatories[controller][i] == signatory) {
            dAppSignatories[controller][i] = dAppSignatories[controller][dAppSignatories[controller].length - 1];
            dAppSignatories[controller].pop();
            break;
        }
    }
}
```

This code does not check that the `signatory` is indeed a signatory for the `controller` in question. Since `removeSignatory()` can be called by arbitrary `signatory` addresses, and since `_addSignatory()` does have extra sanity checks, it would make sense to enforce that the `signatory` actually exists before removing it.

Note: `changeDAppGovernance()` does have this additional check:

```
function changeDAppGovernance(address oldGovernance, address newGovernance) external {
    // ...
    if (!signatories[signatoryKey]) revert AtlasErrors.DAppNotEnabled();
    _removeSignatory(controller, oldGovernance);
    // ...
}
```

Recommendation: Ensure that `signatories[signatoryKey]` is already set, and ensure that the `signatory` exists in the `dAppSignatories[controller]` array. This can be accomplished as follows:

```
  function _removeSignatory(address controller, address signatory) internal {
      bytes32 signatoryKey = keccak256(abi.encodePacked(controller, signatory));
+     if (!signatories[signatoryKey]) revert ...();
      delete signatories[signatoryKey];
      for (uint256 i = 0; i < dAppSignatories[controller].length; i++) {
          if (dAppSignatories[controller][i] == signatory) {
              dAppSignatories[controller][i] =
↪   dAppSignatories[controller][dAppSignatories[controller].length - 1];
              dAppSignatories[controller].pop();
-             break;
+             return;
          }
      }
+     revert();
  }
```

After this change, a check can be removed from `changeDAppGovernance()`:

```
  function changeDAppGovernance(address oldGovernance, address newGovernance) external {
      // ...
-     if (!signatories[signatoryKey]) revert AtlasErrors.DAppNotEnabled();
      _removeSignatory(controller, oldGovernance);
      // ...
  }
```

Fastlane: Solved in PR 240.

Spearbit: Verified.

5.3.21 `ChainlinkAtlasWrapper` sanity check can be stronger

Severity: Low Risk

Context: ChainlinkAtlasWrapper.sol#L110-L117

Description: In the `ChainlinkAtlasWrapper`, the following code determines the median observation, and ensures the observation is a positive value:

```
// Check observations are ordered, then take median observation
for (uint256 i = 0; i < r.observations.length - 1; ++i) {
    bool inOrder = r.observations[i] <= r.observations[i + 1];
    if (!inOrder) revert ObservationsNotOrdered();
}
int192 median = r.observations[r.observations.length / 2];

if (median <= 0) revert AnswerMustBeAboveZero();
```

Since it's enforced that the `median` is a positive value, it is likely that *all* observations should be positive, which is currently not checked.

Recommendation: Consider enforcing that all observations are positive. Since r.observations is sorted, this can be accomplished by checking that r.observations[0] > 0.

Fastlane: Solved in PR 245.

Spearbit: Verified.

5.3.22 Unused DAppOperation fields

Severity: Low Risk

Context: DAppApprovalTypes.sol#L8-L20

Description: The DAppOperation struct has the following definition:

```
struct DAppOperation {
    address from; // signor address
    address to; // Atlas address
    uint256 value;
    uint256 gas;
    uint256 nonce;
    uint256 deadline;
    address control; // control
    address bundler; // msg.sender
    bytes32 userOpHash; // keccak256 of userOp.to, userOp.data
    bytes32 callChainHash; // keccak256 of the solvers' txs
    bytes signature;
}
```

Currently, the to, value, and gas fields are not used in the code (other than to contribute to the hash of the entire struct). As specified in the comments, the to address was likely meant to be checked to match the ATLAS address. The gas and value fields can likely be removed since different structs already cover this functionality.

Recommendation: Add a check that the to in the DAppOperation is equal to ATLAS, and consider removing the gas and value fields.

Fastlane: Solved in PR 176.

Spearbit: Verified.

5.3.23 solverMetaTryCatch() should not have reverting external calls

Severity: Low Risk

Context: ExecutionEnvironment.sol#L142

Description: In both the _getBidAmount() and _executeSolverOperation() functions, a call to solverMetaTryCatch() in the ExecutionEnvironment is made. In both cases, this call may revert, and the error message of this revert has important consequences (e.g. for determining simulated bid amounts, or for assigning blame for the revert). As a result, it's important that there are no external calls in solverMetaTryCatch() that can revert the entire call with an arbitrary error message.

Currently, this is a risk with the call to ERC20(solverOp.bidToken).balanceOf(address(this)), since it is not wrapped in a try-catch block, and solverOp.bidToken can be an arbitrary contract. So, for example, a malicious ERC20 token might revert with the AtlasErrors.BidFindSuccessful() error selector to spoof a fake bid amount. Users will likely not interact with malicious ERC20 implementations anyway, but this poses a risk if they do.

It's also worth noting that several internal reverts can happen in solverMetaTryCatch(), for example, errors with abi.decode() or with arithmetic underflow/overflow. There is less risk in these cases, since these reverts have fixed error selectors and would be treated as a failure (in the case of _getBidAmount()) or as the generic SolverOutcome.EVMError result (in the case of _executeSolverOperation()).

Recommendation: Ensure that all external calls in `solverMetaTryCatch()` are done in a way such that reverts are caught and rethrown with a specific error selector. For the existing `balanceOf()` call, this can be accomplished with a try-catch block, or with a low-level call.

Fastlane: Solved in PR 348.

Spearbit: Verified.

5.3.24 `sessionKeys` can't be expired

Severity: Low Risk

Context: whitepaper

Description: The whitepaper contains:

However ther is no (onchain) functionality to expire or revoke `sessionKeys`.

Recommendation: The keys are inspired by zerodev. This protocol has a way to expire keys: ZeroDevSessionKeyPlugin - revokeSessionKey(). Consider adding something similar.

Fastlane: Sessions keys can be invalidated by having the user submit a transaction that uses the nonce the session key is associated with up. UserOperations also have a deadline, therefore sessionKeys already have an inferred deadline as well. Conclusion: Won't Fix.

Spearbit: Acknowledged.

5.3.25 Workaround `manuallyUpdateNonceTracker()` might not work

Severity: Low Risk

Context: AtlasVerification.sol#L620-L665

Description: The loop in function `getNextNonce()` could run out of gas, although relatively unlikely. A workaround exist via `manuallyUpdateNonceTracker()`. However it is important that the caller is able to call this function. If the caller would be a smart contract, it might not be able to.

```
function getNextNonce(address account, bool sequenced) external view returns (uint256) {
    // ...
    do {
        unchecked { ++n;  }
        bytes32 bitmapKey = keccak256(abi.encode(account, nonceTracker.highestFullAsyncBitmap + n));
        NonceBitmap memory nonceBitmap = nonceBitmaps[bitmapKey];
        bitmap256 = uint256(nonceBitmap.bitmap);
    } while (bitmap256 == FULL_BITMAP);
    // ...
}
function manuallyUpdateNonceTracker(address account) external {
    // ...
}
```

Recommendation: See the suggestion of issue "`Nonce` logic is complicated".

Fastlane: This function has been removed as part of the nonce logic simplification in PR 259.

Spearbit: Verified.

5.3.26 _getMimicCreationCode relies on Solidity format for offsets

Severity: Low Risk

Context: Factory.sol#L160-L206, Mimic.sol

Description: In function _getMimicCreationCode() the mstores are done on very specific locations, recognizable by the statements add(creationCode, 85). So this function relies highly on the compiled solidity code and thus on the exact compiler version and optimization settings. Any updates in these require a change in _getMimicCreationCode() and it is easy to make mistakes.

```
function _getMimicCreationCode(/*...*/) /*...*/ {
    // ...
    creationCode = type(Mimic).creationCode;
    assembly {
        mstore(
            add(creationCode, 85),
            or(
                and(mload(add(creationCode, 85)), not(shl(96,
↪   0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))),
                shl(96, executionLib)
            )
        )
        mstore(
            add(creationCode, 118),
            or(
                and(mload(add(creationCode, 118)), not(shl(96,
↪   0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))),
                shl(96, user)
            )
        )
        mstore(
            add(creationCode, 139),
            or(
                and(
                    mload(add(creationCode, 139)),
                    not(shl(56, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00FFFFFFFF))
                ),
                add(shl(96, controller), add(shl(88, 0x63), shl(56, callConfig)))
            )
        )
        mstore(add(creationCode, 165), controlCodeHash)
    }
}
```

Recommendation: See the suggestion for the issue "Mimic can be optimized" which will eliminate this issue.

Fastlane: A potential solution increases gas cost by about 11 000 per metacall, so we have decided to not merge it. The Solidity compiler version dependency is okay for now, as we can see quite clearly when it breaks in the tests.

Spearbit: Acknowledged.

5.3.27 `_getMimicCreationCode` relies on Solidity format for layout

Severity: Low Risk

Context: Factory.sol#L160-L206, Mimic.sol

Description: In function `_getMimicCreationCode()`, the `add(shl(88, 0x63) ..)` is redundant, because its already kept by the mask with `00` above.

With the applied compiler version, this value in the `Mimic` code is `0x63`. In that case, `OR`-ing it with `0x63` results in the same value. However it costs additional gas and with other Solidity versions this value might change.

```
function _getMimicCreationCode(/*...*/) /*...*/ {
        // ...
        mstore(
            add(creationCode, 139),
            or(
                and(
                    mload(add(creationCode, 139)),
                    not(shl(56, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00FFFFFFFF))  // note the 00
                ),
                add(shl(96, controller), add(shl(88, 0x63), shl(56, callConfig)))
            )
        )
        mstore(add(creationCode, 165), controlCodeHash)
    }
}
```

Recommendation: See the suggestion for the issue "`Mimic` can be optimized" which will eliminate this issue.

Fastlane: Solved in PR 286.

Spearbit: Verified.

5.3.28 Return a tuple of `(preOpsReturnData, userReturnData)` in `_preOpsUserExecutionIteration()`

Severity: Low Risk

Context: Atlas.sol#L225

Description: `returnData` returned by `_preOpsUserExecutionIteration()` means different things in different context:

needsPreOpsCall && needsPreOpsReturnData	needsUserReturnData	returnData
F	F	empty
F	T	userReturnData
T	F	preOpsReturnData
T	T	preOpsReturnData++userReturnData

We modified this function a bit to test difference cases.

For this case:

- `needsPreOpsCall` = T, `needsPreOpsReturnData` = F, `needsUserReturnData` = F,

even if the function returns `preOpsReturnData` (instead of empty), the test cases pass. That indicates either a lack of coverage or a bug in the code.

Recommendation: Return both `preOpsReturnData` and `userReturnData` (empty or full depending on these booleans) in a tuple or a 2-sized array. It also reduces many branches in this function.

Fastlane: Solved in PR 227 and PR 260. Only 1 return data between preOps and user call in a single metacall.

Spearbit: Verified.

5.3.29 `WETH_X_GOVERNANCE_POOL` may not have governance token

Severity: Low Risk

Context: V2DAppControl.sol#L78-L83

Description: The following check doesn't protect against the case when governance token isn't part of the pool. It only ensures that at least one of the tokens is WETH.

```
govIsTok0 = (IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token0() == GOVERNANCE_TOKEN);
if (govIsTok0) {
    require(IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token1() == WETH, "INVALID TOKEN PAIR");
} else {
    require(IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token0() == WETH, "INVALID TOKEN PAIR");
}
```

Recommendation: In the `else` condition, check that `token1` is `GOVERNANCE_TOKEN`:

```
  if (govIsTok0) {
      require(IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token1() == WETH, "INVALID TOKEN PAIR");
  } else {
      require(IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token0() == WETH, "INVALID TOKEN PAIR");
+     require(IUniswapV2Pair(WETH_X_GOVERNANCE_POOL).token1() == GOVERNANCE_TOKEN, "INVALID TOKEN
↪   PAIR");
  }
```

Fastlane: Solved in PR 237.

Spearbit: Verified.

5.3.30 `metacall()` doesn't always use `netGasSurcharge`

Severity: Low Risk

Context: Atlas.sol#L42-L96

Description: In function `metacall()`, when a `solver` has won the auction then a `netGasSurcharge` is withheld and an `emit` is done. When an error occurs, then `netGasSurcharge` isn't withheld and no `emit` is done, even if `metacall()` doesn't revert itself. Adding a `netGasSurcharge` might be useful to prevent spam transactions.

```
function metacall(/*...*/) /*...*/ {
    // ...
    try this.execute{ value: msg.value }(/*...*/)
    returns (/*...*/) {
        (uint256 ethPaidToBundler, uint256 netGasSurcharge) = _settle({ /*...*/ });
        emit MetacallResult(/*...*/, ethPaidToBundler, netGasSurcharge);
        );
    } catch (bytes memory revertData) {
        if (msg.value != 0) SafeTransferLib.safeTransferETH(msg.sender, msg.value); // send to bundler
       // no netGasSurcharge
       // no emit
    }
    // ...
}
```

Recommendation: When an error has occured, still consider to withhold a `netGasSurcharge` and/or do an `emit`.

Fastlane: Solved in PR 343 by emitting an event. We don't need to take a surcharge in the case of an `execute()` failure - the gas cost of the failing tx should be disincentive enough.

Spearbit: Verified.

5.3.31 Use `tryRecover()` for signature verification

Severity: Low Risk

Context: AtlasVerification.sol#L266, AtlasVerification.sol#L504, AtlasVerification.sol#L604, AtlETH.sol#L142, ChainlinkDAppControl.sol#L136-L162

Description: `recover()` reverts if the recovered signature is `address(0)` which is the case for invalid signatures.

So call validation reverts instead of bubbling up the error. These function is called via `metacall()` which isn't supposed to revert as per this comment:

```
// Gracefully return if not valid. This allows signature data to be stored, which helps prevent
// replay attacks.
// NOTE: Currently reverting instead of graceful return to help w/ testing. TODO - still reverting?
(bytes32 userOpHash, ValidCallsResult validCallsResult) =
↪   IAtlasVerification(VERIFICATION).validateCalls(
    dConfig, userOp, solverOps, dAppOp, msg.value, msg.sender, isSimulation
);
```

A single invalid signature from solver can revert the entire execute and prevent `userOp` to be executed. Although this invalid signature shouldn't land onchain for actual execution as it should be caught in simulation.

ERC-4337 has the same requirement for signature failures.

Note: some of the example use `ecrecover()`. This returns 0 when the signatures don't match which might go undetected.

Recommendation: Using `tryRecover()` which returns error code when the retrieved address is `address(0)`. Thus, default revert can be avoided and the execution can gracefully return.

In some cases like `AtlETH.permit()`, you may want to keep it as-is to avoid deviating from the source (Solmate ERC20) too much.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.3.32 Disposable `sessionKey`s might be deleted too soon

Severity: Low Risk

Context: Atlas.sol#L42-L96

Description: The bundler can be a (disposable) `sessionKey`, which can receive `ETH` from `metacall()`. After this `sessionKey` is disposed of, the ETH is no longer available.

```
function metacall(/*...*/) /*...*/ {
    // ...
    // Refund the msg.value to sender if it errored
    if (msg.value != 0) SafeTransferLib.safeTransferETH(msg.sender, msg.value);
    // ...
}
```

Recommendation: Document that `ETH` must be removed from the `sessionKey` account before deleted it. *Note: This remains relevant even when this suggestion is followed: "Call to `safeTransferETH` can do unwanted actions".*

Fastlane: Solved in PR 266.

Spearbit: Verified.

5.3.33 OR operator is used instead of AND operator

Severity: Low Risk

Context: SwapIntent.sol#L100-L121

Description: `SwapIntent.swap()` function does a require check and then uses the same condition in the `if` condition:

```
require(swapIntent.tokenUserSells != swapIntent.auctionBaseCurrency, "ERR-PI008 SellIsSurplus");
// ...
if (
    swapIntent.auctionBaseCurrency != swapIntent.tokenUserSells
        || swapIntent.auctionBaseCurrency != swapIntent.tokenUserBuys
) {
```

Regardless of the `require` check, `(a != c || a != b)` is always `true`. This `if` condition is meant to be an AND instead of OR.

Recommendation: The first condition can be removed entirely since the preceding `require` check ensures it's always `true`:

```
  if (
-     swapIntent.auctionBaseCurrency != swapIntent.tokenUserSells
-         || swapIntent.auctionBaseCurrency != swapIntent.tokenUserBuys
+     swapIntent.auctionBaseCurrency != swapIntent.tokenUserBuys
  ) {
```

Fastlane: Solved due to refactoring.

Spearbit: Verified.

5.3.34 Calls to `AtlETH` functions not restricted

Severity: Low Risk

Context: AtlETH.sol

Description: Several functions of `AtlETH` could potentially be called during a `metacall()`, possibly while execution is given to another contract with a `safeTransferETH()` call. This could interfere with the functionality of Atlas which relies on the `AtlETH` information to stay the same.

Also see the issue "Call to `safeTransferETH` can do unwanted actions".

Recommendation: Consider checking if `Atlas` is in an `UNLOCKED` state at the top of any sensitive `AtlETH` functions. Consider the following functions for this check:

- `deposit()`
- `withdraw()`
- `transfer()`
- `transferFrom()`
- `bond()`
- `depositAndBond()`
- `unbond()`
- `redeem()`
- `withdrawSurcharge()`

Fastlane: Solved in PR 340.

Spearbit: Verified.

5.3.35 Call to `safeTransferETH` can do unwanted actions

Severity: Low Risk

Context: Atlas.sol#L91, GasAccounting.sol#L295

Description: `metacall()` and `_settle()` do an `safeTransferETH()` to the bundler. The bundler can intercept this call via an `receive()` and do one of the following unwanted actions:

- Do `revert()`. In that case the original call will also revert, see Solmate SafeTransferLib. This way for example the transactions of specific solvers could be reverted. Note: When `reuseUserOp == false` then `reverts` should be avoided.
- Call `reconcile()`, see issue "`Reconcile()` can be called by anyone".
- Call functions of `AtlETH`, see issue "Calls to `AtlETH` functions not restricted".

```
function metacall(/*...*/) /*...*/ {
     // ...
    try this.execute{ value: msg.value }(/*...*/)
        // ...
    } catch (bytes memory revertData) {
        // ...
        if (msg.value != 0) SafeTransferLib.safeTransferETH(msg.sender, msg.value); // bundler
    }
    // ...
}
function _settle(/*...*/) /*...*/ {
    // ...
    SafeTransferLib.safeTransferETH(bundler, _claims);
    // ...
}
```

Also, note that these scenarios would give the bundler control flow *after* the final checks on `deposits/withdrawals/claims`, but *before* the main Atlas lock is released. This is a potentially dangerous location to give control flow, as any fund transfers would be untracked. Fortunately, this is not exploitable because, for example, the `ExecutionEnvironment` currently can't reach a `delegatecall` without entering `Atlas` first. However, eliminating this dangerous external call could help prevent future issues if the code is changed.

Recommendation: Consider adding the funds to the `AtlETH` balance instead of sending them. This is also known as the `pull over push` pattern. *Note: if there is no winning `solver` then `_settle()` uses the `bunder` to `_assign()` the cost so the `bunder` should have `AtlETH` balances anyway.*

Fastlane: Largely solved due to refactoring. We now just have 1 line at the end of _settle() which transfers ETH directly to bundler:

```
SafeTransferLib.safeTransferETH(ctx.bundler, claimsPaidToBundler);
```

While this gives control flow to the bundler, we think this is safe because:

- `reconcile()` cannot be called at this point.
- The ERC20 functions of AtlETH have been removed, so those are not an attack vector anymore.
- The bundler would still be paying gas for the entire metacall tx, so there is an economic disincentive to not revert. Atlas is still in a locked state as `_releaseAccountingLock()` is only called right at the end of the metacall transaction.

Furthermore, the UX benefit of sending the refunded ETH directly to the bundler's address are significant over the friction of additional calls to unbond/redeem any credited AtlETH.

Spearbit: Verified.

5.4 Gas Optimization

5.4.1 Include simulation mode information as custom error argument

Severity: Gas Optimization

Context: Atlas.sol#L146-L147,

Description: wherever `isSimulation` is used, the code is always reverting except at Atlas.sol#L133-L135 where the execution can still continue.

To remove the `if/else` branch on reverts conditioned on `isSimulation` - a custom error, which takes `isSimulation` as an argument, can be used. It simplifies the code.

Recommendation: Consider updating all the if/else branches which revert based on `isSimulation` as follows:

```
- if (key.isSimulation) revert PostOpsSimFail();
- else revert PostOpsFail();
+ revert PostOpsFail(key.isSimulation)
```

Fastlane: Decided not fixing.

Spearbit: Acknowledged.

5.4.2 `_deduct()` reverts can be improved

Severity: Gas Optimization

Context: AtlETH.sol#L221-L243

Description: Function `_deduct()` reverts with a generic error if `aData.unbonding < _shortfall`, while in other situations it has a specific error. The revert `InsufficientBalanceForDeduction()` could get the balance in a cheaper way.

```
function _deduct(address account, uint256 amount) internal {
    uint112 amt = uint112(amount);
    EscrowAccountBalance memory aData = _balanceOf[account];
    uint112 balance = aData.balance;
    if (amt <= balance) {
        // ...
    } else if (block.number > accessData[account].lastAccessedBlock + ESCROW_DURATION) {
        uint112 _shortfall = amt - balance;
        // ...
        aData.unbonding -= _shortfall; // underflow here to revert if insufficient balance
        // ...
    } else {
        revert InsufficientBalanceForDeduction(_balanceOf[account].balance, amount);
    }
}
```

Recommendation: Consider to have a specific error message in case the `unbonding` balance is insufficient. Consider changing the code to:

```
- revert InsufficientBalanceForDeduction(_balanceOf[account].balance, amount);
+ revert InsufficientBalanceForDeduction(uint256(balance), amount);
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.3 Code duplication in `initializeGovernance()`

Severity: Gas Optimization

Context: DAppIntegration.sol#L57-L72, DAppIntegration.sol#L140-L145

Description: `initializeGovernance()` function duplicates the code of `_addSignatory()`. The only difference is the error message.

```
function initializeGovernance(address controller) external {
    // ...
    // Add DAppControl gov as a signatory
    bytes32 signatoryKey = keccak256(abi.encodePacked(controller, msg.sender));
    if (signatories[signatoryKey]) revert AtlasErrors.OwnerActive();
    signatories[signatoryKey] = true;
    dAppSignatories[controller].push(msg.sender);
    ...
}
function _addSignatory(address controller, address signatory) internal {
    bytes32 signatoryKey = keccak256(abi.encodePacked(controller, signatory));
    if (signatories[signatoryKey]) revert AtlasErrors.SignatoryActive();
    signatories[signatoryKey] = true;
    dAppSignatories[controller].push(signatory);
}
```

Recommendation: Condider calling `_addSignatory()` from `initializeGovernance()`.

Fastlane: Solved in PR 215.

Spearbit: Verified.

5.4.4 `ExecutionBase` functions contain redundant checks

Severity: Gas Optimization

Context: ExecutionBase.sol#L198-L224, GasAccounting.sol#L41-L55, Permit69.sol#L56-L107

Description: The functions `_contribute()`, `_borrow()`, `_transferUserERC20()` and `_transferDAppERC20()` check `msg.sender == atlas` however the value of this is limited. The real access control check is on the receiving side, e.g. in `contribute()`, `borrow()`, `transferUserERC20()` and `transferDAppERC20()`.

```
function _contribute(uint256 amt) internal {
    if (msg.sender != atlas) revert AtlasErrors.OnlyAtlas();
    // ...
    IEscrow(atlas).contribute{ value: amt }();
}
function _borrow(uint256 amt) internal {
    if (msg.sender != atlas) revert AtlasErrors.OnlyAtlas();
    IEscrow(atlas).borrow(amt);
}
function _transferUserERC20(address token, address destination, uint256 amount) internal {
    if (msg.sender != atlas) {
        revert AtlasErrors.OnlyAtlas();
    }
    IPermit69(atlas).transferUserERC20(token, destination, amount, _user(), _control(), _config(),
↪   _lockState());
}
function _transferDAppERC20(address token, address destination, uint256 amount) internal {
    if (msg.sender != atlas) {
        revert AtlasErrors.OnlyAtlas();
    }
    IPermit69(atlas).transferDAppERC20(token, destination, amount, _user(), _control(), _config(),
↪   _lockState());
}
```

Functions in `Atlas`:

```
function contribute() external payable {
    if (lock != msg.sender) revert InvalidExecutionEnvironment(lock);
    // ...
}
function borrow(uint256 amount) external payable {
    if (lock != msg.sender) revert InvalidExecutionEnvironment(lock);
    // ...
}
function transferUserERC20(...) ... {
    _verifyCallerIsExecutionEnv(user, controller, callConfig);
    // ...
}
function transferDAppERC20(...) ... {
    _verifyCallerIsExecutionEnv(user, controller, callConfig);
    // ...
}
```

Recommendation: Consider removing the checks in `_contribute()`, `_borrow()`, `_transferUserERC20()` and `_transferDAppERC20()`.

Fastlane: Solved in PR 214.

Spearbit: Verified.

5.4.5 `latestTimestamp()` can be optimized

Severity: Gas Optimization

Context: ChainlinkAtlasWrapper.sol#L53-L59

Description: Function `latestTimestamp()` calls `BASE_FEED.latestTimestamp()` twice. It also accesses the `storage` variable `atlasLatestTimestamp` twice, which is relatively expensive. This can be optimized.

```
uint256 public atlasLatestTimestamp;
function latestTimestamp() public view returns (uint256) {
    if (BASE_FEED.latestTimestamp() >= atlasLatestTimestamp) {
        return BASE_FEED.latestTimestamp(); // second call
    } else {
        return atlasLatestTimestamp; // second access
    }
}
```

Recommendation: Consider storing the result of `BASE_FEED.latestTimestamp()` and the value of `atlasLatestTimestamp` in a temporary variable.

Fastlane: Solved in PR 206.

Spearbit: Verified.

5.4.6 `_getSortingData()` can be optimized

Severity: Gas Optimization

Context: Sorter.sol#L59-L131

Description: `_verifySolverEligibility()` is called in a loop and every time calls `getUserOperationHash()`. As `getUserOperationHash()` is a relative expensive function and the input is always the same, it would be cheaper to do this outside the loop.

```
function _getSortingData(/*...*/) /*...*/ {
    // ...
    for (; i < count;) {
        if (/*...*/ && _verifySolverEligibility(dConfig, userOp, solverOps[i])) {
            // ...
        } else {
            // ...
        }
        // ...
    }
    // ...
}
function _verifySolverEligibility(/*...*/) /*...*/ {
    // ...
    bytes32 userOpHash = CallVerification.getUserOperationHash(userOp);
    // ...
}
```

Recommendation: Consider moving the call to `getUserOperationHash()` outside the loop.

Fastlane: Solved in PR 209.

Spearbit: Verified.

5.4.7 `sortBids()` can be optimized

Severity: Gas Optimization

Context: Sorter.sol#L24-L53

Description: Function `sortBids()` calculates `count -invalid` twice. This can be optimized, while also increases readability.

```
function sortBids(/*...*/) /*...*/ {
    // ...
    SolverOperation[] memory solverOpsSorted = new SolverOperation[](count - invalid);
    count -= invalid;
    // ...
}
```

Recommendation: Consider changing the code to:

```
  function sortBids(...) ... {
      // ...
-     SolverOperation[] memory solverOpsSorted = new SolverOperation[](count - invalid);
      count -= invalid;
+     SolverOperation[] memory solverOpsSorted = new SolverOperation[](count);
      // ...
  }
```

Fastlane: Solved in PR 185.

Spearbit: Verified.

5.4.8 `factoryWithdrawERC20()` and `factoryWithdrawEther()` are unused

Severity: Gas Optimization

Context: ExecutionEnvironment.sol#L367-L377, ExecutionEnvironment.sol#L399-L409

Description: The functions `factoryWithdrawERC20()` and `factoryWithdrawEther()` are not called from `Atlas`.

Note: they could be called via a `userOp` in combination with `delegatecall` and this issue: "`validControl` / `onlyAtlasEnvironment` are not effective in `delegatecall` situation".

This poses no extra risk because the `ExecutionEnvironment` call already can access the funds.

```
function factoryWithdrawERC20(address msgSender, address token, uint256 amount) external {
    require(msg.sender == atlas, "ERR-EC10 NotFactory");
    require(msgSender == _user(), "ERR-EC11 NotEnvironmentOwner");
    require(ISafetyLocks(atlas).isUnlocked(), "ERR-EC15 EscrowLocked");
    if (ERC20(token).balanceOf(address(this)) >= amount) {
        SafeTransferLib.safeTransfer(ERC20(token), _user(), amount);
    } else {
        revert("ERR-EC02 BalanceTooLow");
    }
}
function factoryWithdrawEther(address msgSender, uint256 amount) external {
    require(msg.sender == atlas, "ERR-EC10 NotFactory");
    require(msgSender == _user(), "ERR-EC11 NotEnvironmentOwner");
    require(ISafetyLocks(atlas).isUnlocked(), "ERR-EC15 EscrowLocked");
    if (address(this).balance >= amount) {
        SafeTransferLib.safeTransferETH(_user(), amount);
    } else {
        revert("ERR-EC03 BalanceTooLow");
    }
}
```

Recommendation: Consider removing the functions `factoryWithdrawERC20()` and `factoryWithdrawEther()`, although they might be useful again when fixing issue "Locking mechanism is complicated - links have to added".

Fastlane: Solved in PR 168 and PR 357.

Spearbit: Verified.

5.4.9 Moving `validateBalances()` to `Atlas`

Severity: Gas Optimization

Context: ExecutionEnvironment.sol#L142-L317

Description: Currently there is some back and forth calling between the `ExecutionEnvironment`, the `Solver` and `Altas`. This is complicated, has overhead and is potentially risky.

```
function solverMetaTryCatch(
    // Execute the solver call.
    // which calls `IEscrow(_atlas).reconcile`

    if (endBalance > 0) {
        IEscrow(atlas).contribute{ value: endBalance }();
    }
    (, success) = IEscrow(atlas).validateBalances();
    if (!success) {
        revert AtlasErrors.BalanceNotReconciled();
    }
}
```

Recommendation: Consider calling `validateBalances()` from `Atlas` itself.

Also see issues:

- "`validateBalances()` and `_checkAtlasIsUnlocked()` could use `isUnlocked()`".
- "`Borrow()`s after `validateBalances()`".
- "Locking mechanism is complicated".

Fastlane: Solved in PR 225. Atlas calls directly to solver. Solver must repay via `reconcile()`. Then instead of `validateBalances()` (which is now removed), we check solver has repaid at the end of `solverCall()` using `_solverLockData()`.

Spearbit: Verified.

5.4.10 `_executeSolverOperation()` executes the same line twice

Severity: Gas Optimization

Context: Escrow.sol#L97-L168

Description: In `_executeSolverOperation()` the statement `key.solverOutcome = uint24(result)` might be done twice in certain situations.

```
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.canExecute()) {
        // ...
        if (result.canExecute() && _trySolverLock(solverOp)) {
            // ...
            key.solverOutcome = uint24(result);
            if (result.executionSuccessful()) {
                // ...
                return (true, key);
            }
        }
    }
    key.solverOutcome = uint24(result); // has been done before in certain situations
    // ...
    return (false, key);
}
```

Recommendation: Consider moving the first `key.solverOutcome = uint24(result)` directly before the `return`.

```
  function _executeSolverOperation(...) ... {
      // ...
      if (result.canExecute()) {
          // ...
          if (result.canExecute() && _trySolverLock(solverOp)) {
              // ...
-            key.solverOutcome = uint24(result);
              if (result.executionSuccessful()) {
                  // ...
+                 key.solverOutcome = uint24(result);
                  return (true, key);
              }
          }
      }
      key.solverOutcome = uint24(result);
      // ...
      return (false, key);
  }
```

Fastlane: Solved in PR 181.

Spearbit: Verified.

5.4.11 Assign with or operator (|=) can be reduced

Severity: Gas Optimization

Context: Escrow.sol#L132-L140, Escrow.sol#L279

Description: In function `_executeSolverOperation()` `result` is verified to be `0` before calling `_solverOpWrapper()`, so the or operator (`|`) is not necessary.

```
 function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.canExecute() && _trySolverLock(solverOp)) { // result is now 0
        // ...
        result |= _solverOpWrapper(...);  // |  not necessary
        // ...
    }
    // ...
}
```

In `_validateSolverOperation()` the value is returned directly so no need to first assign it to `result`.

```
function _validateSolverOperation(/*...*/) /*...*/ {
    // ...
    return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit); // = not necessary
    // ...
}
```

Recommendation: Consider changing the code as follows:

```
- result |= _solverOpWrapper(...);
+ result =  _solverOpWrapper(...);
```

```
- return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
+ return (result |  1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
```

Fastlane: Solved in PR 182.

Spearbit: Verified.

5.4.12 Parameter of `manuallyUpdateNonceTracker()` not necessary

Severity: Gas Optimization

Context: AtlasVerification.sol#L646-L665

Description: Function `manuallyUpdateNonceTracker()` enforces the parameter `account` to be equal to `msg.sender`. In that case supplying `account` isn't necessary.

```
function manuallyUpdateNonceTracker(address account) external {
    if (msg.sender != account) revert AtlasErrors.OnlyAccount();
    // ...
}
```

Recommendation: Consider changing the code to:

```
- function manuallyUpdateNonceTracker(address account) external {
+ function manuallyUpdateNonceTracker() external {
-    if (msg.sender != account) revert AtlasErrors.OnlyAccount();
     // ...
  }
```

And replace all occurances of `account` with `msg.sender`.

Fastlane: Solved in PR 259. This function has been removed as part of the nonce logic simplification.

Spearbit: Verified.

5.4.13 `Nonce` logic is complicated

Severity: Gas Optimization

Context: AtlasVerification.sol#L386-L447

Description: The logic to handle async `nonces` is rather complicated and thus relative gas intensive. This is mainly done to be able to do `getNextNonce()` onchain. This approach is not foul proof either because one of the `nonces` might already be in transit without any onchain updates.

Also see issues:

- Workaround `manuallyUpdateNonceTracker()` might not work.
- Function `manuallyUpdateNonceTracker()` can miss blocks that are not completely filled.

One example:

```
function _handleNonces(address account, uint256 nonce, bool async, bool isSimulation) internal returns
↪   (bool) {
    // ...
    // ASYNC NONCES
    uint256 bitmapIndex = ((nonce - 1) / 240) + 1;
    uint256 bitmapNonce = ((nonce - 1) % 240);
    bytes32 bitmapKey = keccak256(abi.encode(account, bitmapIndex));
    NonceBitmap memory nonceBitmap = nonceBitmaps[bitmapKey];
    uint256 bitmap = uint256(nonceBitmap.bitmap);
    if (_nonceUsedInBitmap(bitmap, bitmapNonce)) { return false; }
    // ...
    bitmap |= 1 << bitmapNonce;
    nonceBitmap.bitmap = uint240(bitmap);
    if (bitmapNonce + 1 > uint256(nonceBitmap.highestUsedNonce)) {
        nonceBitmap.highestUsedNonce = uint8(bitmapNonce + 1);
    }
    if (bitmap == FULL_BITMAP) {
        if (bitmapIndex == nonceTracker.highestFullAsyncBitmap + 1) {
            nonceTracker = _incrementHighestFullAsyncBitmap(nonceTracker, account);
        }
    }
    nonceBitmaps[bitmapKey] = nonceBitmap;
    // ...
}
```

Recommendation: Consider simplifying the `nonce` logic for example modeled after the code in Permit2.

Move the complicated logic of `getNextNonce()` offchain. If an async `nonce` needs to be allocated onchain consider using a random `nonce` for example using `block.prevrandao`.

- ***Note 1:*** *this is not perfectly random but for this case it should be sufficient.*
- ***Note 2:*** *the `storage` of the `nonce` will be less efficient because `bitmaps` will not be filled.*

See additional links for inspiration:

- Permit2-nonce-finder.
- Permit2 signatures-on-the-frontend.
- Brink available nonces.

Fastlane: Solved in PR 259.

Spearbit: Verified.

5.4.14 `userWrapper()` considers entire balance instead of `msg.value`

Severity: Gas Optimization

Context: ExecutionEnvironment.sol#L99, Escrow.sol#L79

Description: `userWrapper()` is called such that `msg.value == userOp.value`:

```
(success, userData) = environment.call{ value: userOp.value }(userData);
```

However, `userWrapper()` validates `userOp.value` against `address(this).balance`:

```
require(address(this).balance >= userOp.value, "ERR-CE01 ValueExceedsBalance");
```

`ExecutionEnvironment` can have some `ETH` balance already since it as `receive()` function. Thus, `address(this).balance >= msg.value`. Reading `address(this).balance` is more expensive operation than reading `msg.value`.

Thus, checking against `msg.value` is technically more accurate and also cheaper.

Recommendation: Update as:

```
- require(address(this).balance >= userOp.value, "ERR-CE01 ValueExceedsBalance");
+ require(msg.value >= userOp.value, "ERR-CE01 ValueExceedsBalance");
```

Fastlane: Solved in PR 244.

Spearbit: Verified.

5.4.15 `netGasSurcharge` is declared twice

Severity: Gas Optimization

Context: GasAccounting.sol#L289

Description: `netGasSurcharge` is a named return variable, but it's declared again later.

```
function _settle(/*...*/) /*...*/ returns (/*...*/, uint256 netGasSurcharge)  {
   // ...
   uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000; // declared again
   // ...
}
```

Recommendation: Update as:

```
- uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;
+ netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.16 Cache variables

Severity: Gas Optimization

Context: ChainlinkDAppControl.sol#L153, ChainlinkDAppControl.sol#L219, ChainlinkDAppControl.sol#L206-L207, SwapIntent.sol#L146-L150

Description: Highlighted areas are where variables can be cached to avoid reading from storage more than once.

- ChainlinkDAppControl.sol#L153: `verificationVars[baseChainlinkFeed]` can be extracted in a storage variable to avoid getting its value in each loop.
- ChainlinkDAppControl.sol#L219: `signers.length` is read for each iteration.
- ChainlinkDAppControl.sol#L206-L207: last signer is read twice.
- SwapIntent.sol#L146-L150: `swapIntent.conditions.length` is read twice.

Recommendation:

- ChainlinkDAppControl.sol#L153: Update to:

```
  VerificationVars storage verificationVar = verificationVars[baseChainlinkFeed];
  for (uint256 i = 0; i < rs.length; ++i) {
      address signer = ecrecover(reportHash, uint8(rawVs[i]) + 27, rs[i], ss[i]);
      currentOracle = verificationVar.oracles[signer];
```

- ChainlinkDAppControl.sol#L219: Cache `signers.length`.
- ChainlinkDAppControl.sol#L206-L207: Update to:

```
address lastSigner = signers[signers.length - 1];
signers[oracle.index] = lastSigner;
verificationVars[baseChainlinkFeed].oracles[lastSigner].index = oracle.index;
```

- SwapIntent.sol#L146-L150: Use `maxUserConditions` in the `require` check too.

Fastlane: Solved in PR 184.

Spearbit: Verified.

5.4.17 No need to check for signature length

Severity: Gas Optimization

Context: AtlasVerification.sol#L503, AtlasVerification.sol#L603

Description: there's no need for this check:

```
if (dAppOp.signature.length == 0) return false;
```

```
if (userOp.signature.length == 0) return false;
```

`recover()` does this check already. If `tryRecover()` is used instead as suggested in the issue "Use `tryRecover()` for signature verification" is followed`tryRecover()`, it won't throw on invalid signature.

Recommendation: Remove these checks and use `tryRecover()`.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.18 Only one `...get...Hash()` function uses `calldata`

Severity: Gas Optimization

Context: AtlasVerification.sol#L481-L497, AtlasVerification.sol#L580-L597, AtlasVerification.sol#L273-L291, CallVerification.sol#L11-L17

Description: Function `_getSolverHash()` uses a `calldata`, but other comparable functions use `memory`, which costs more gas. All functions could use `calldata`.

```
function _getSolverHash(SolverOperation calldata solverOp) /*...*/ { }
function _getProofHash(DAppOperation memory approval) /*...*/ { }
function _getProofHash(UserOperation memory userOp) /*...*/ { }
function getUserOperationHash(UserOperation memory userOp) /*...*/ { }
function getAltOperationHash(UserOperation memory userOp) /*...*/ { }
```

Recommendation: Consider using `calldata` in all functions.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.19 Async vs sequential vs sequenced

Severity: Gas Optimization

Context: AtlasVerification.sol#L307-L447, AtlasVerification.sol#L532-L575

Description: There are multiple terms to indicate the same concept, which can be confusing:

- async
- sequential
- sequenced

Especially combined with negating (`!`) the values, the risk for confusion increases.

```
function _verifyDApp(
     // ...
    if (!_handleNonces(..., !dConfig.callConfig.needsSequencedDAppNonces(), ...)) { // uses !
        // ...
    }
    // ...
}
function _verifyUser(
    // ...
    if (!_handleNonces(..., !dConfig.callConfig.needsSequencedUserNonces(), ...)) { // uses !
        // ...
    }
    // ...
}
function _handleNonces(/*...*/, bool async, /*...*/) internal returns (bool) {
    // ...
    if (!async) { // uses !
        // SEQUENTIAL NONCES
       // ...
    } else {
        // ASYNC NONCES
        // ...
    }
}
/// @param sequenced A boolean indicating if the nonce should be sequential (true) or async (false).
function getNextNonce(..., bool sequenced) external view returns (uint256) {
    // ...
}
function needsSequencedUserNonces(...) internal pure returns (bool sequenced) {
    // ...
}
```

Recommendation: Consider to standardize on one term, preferably a term that doesn't require negation. For example `sequenced`. This will even safe some gas.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.20 Checks in function _verifyDApp() can be simplified

Severity: Gas Optimization

Context: AtlasVerification.sol#L307-L377

Description: Function _verifyDApp() defines and assigns the variable bypassSignatoryCheck twice with the same value. This can be optimized. The first time the variable isn't used, which means simulation mode isn't handled optimally. This might allow transactions to be bundled that waste gas.

```
function _verifyDApp(
    // ...
    bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0); // not used
    if (!isSimulation) { // should probably be bypassSignatoryCheck
        return (false, ValidCallsResult.InvalidBundler);
    }
    // ...
    bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0); // same value as above
    if (!bypassSignatoryCheck) {
        return (false, ValidCallsResult.DAppSignatureInvalid);
    }
    // ...
    if (dAppOp.from == address(0) && isSimulation) {
            return (true, ValidCallsResult.Valid);
    }
}
```

The situation where dAppOp.from == address(0) would be true only for simUserOperation() calls via the Simulator (because these sims may be done before a dAppOp is available, so will not be able to check e.g. dapp sig or dapp nonce) and not for simulations involving solverOp(s), as those take a dAppOp param, so those dAppOp properties can be checked.

Recommendation: Consider combining the checks in one variable:

```
  function _verifyDApp(
      // ...
+     bool skipDAppOpChecks = isSimulation && dAppOp.from == address(0);
-     bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0);
-      if (!isSimulation) {
+      if (!skipDAppOpChecks ) {
          return (false, ValidCallsResult.InvalidBundler);
      }
      // ...
-     bool bypassSignatoryCheck = isSimulation && dAppOp.from == address(0);
-     if (!bypassSignatoryCheck) {
+     if (!skipDAppOpChecks ) {
          return (false, ValidCallsResult.DAppSignatureInvalid);
      }
      // ...
-     if (dAppOp.from == address(0) && isSimulation) {
+     if (skipDAppOpChecks ) {
              return (true, ValidCallsResult.Valid);
      }
  }
```

Also see issue "Simulation code path can be kept off-chain".

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.21 `Mimic` can be optimized

Severity: Gas Optimization

Context: ExecutionEnvironment.sol#L31-L33, Mimic.sol, ExecutionBase.sol#L116-L120, ExecutionEnvironment.sol#L75

Description: Any call to `ExecutionEnvironment` goes through `Mimic` which appends `userOp.from`, `control`, call `config`, `control.codehash` to the calldata:

```
0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa is standin for the ExecutionEnvironment, which is a de facto
↪  library
0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB is standin for the user's EOA address
0xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCcccccccC is standin for the dApp control address
0x2222 is standin for the call configuration
0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee is the dApp control contract's
↪  .codehash
```

This is to ensure that a unique `ExecutionEnvironment` is deployed for this combination and also to verify that correct parameters are passed to any external call to `ExecutionEnvironment`.

Including these parameters in the create2 salt ensures the uniqueness of deployment address. Also, since `ExecutionEnvironment` can only be called from `atlas`, we can be sure that correct parameters are passed assuming correct code. Thus, the verification steps like the following can be skipped:

```
if (userOp.from != _user()) {
    revert("ERR-CE02 InvalidUser");
}
```

Here, `_user()` parses user address from the appended calldata.

Thus, following this logic, we can remove appending `userOp.from`, `control`, call `config` to the calldata as they all can be retrieved from function arguments already.

`control.codehash` is used for the following check:

```
modifier validControlHash() {
    if (_control().codehash != _controlCodeHash()) {
        revert("ERR-EV008 InvalidCodeHash");
    }
    _;
}
```

This check is to account for the possibility of changing the code via selfdestruct? With Dencun upgrade, selfdestruct can destroy the code only when called in the creation tx (rollups and other EVM chains haven't upgraded to Dencun yet). However, there are cases where this check doesn't provide any protection against:

- If `control` is a proxy, the implementation can change without changing its `codehash`.
- `control` can change its execution without using proxy by detecting which stage Atlas is in. By calling `solverMetaTryCatch` and checking it's in lock state or not. Although, with this malicious behavior, it likely won't be used by an honest user or an honest dapp.
- `control` can also change its behavior based on its state which isn't included in `codehash`.

Also see:

- Function `_getMimicCreationCode` relies on Solidity format for offsets
- Function `_getMimicCreationCode` relies on Solidity format for layout

Recommendation: Consider deploying the `ExecutionEnvironment` in the following way:

- Use `Clones` proxy.

- Pass `config` as a new `uint32` argument to all the functions, or retrieve it via `IDAppControl(controller).CALL_CONFIG();`.
- Remove corresponding verification checks in `ExecutionEnvironment`:
  - `validControlHash()`.
  - the check against `_user()`.
- Include the following to the salt for `create2` to ensure address uniqueness:
  - User address.
  - Control address.
  - Call `config`.
- For functions that are directly called on `ExecutionEnvironment` like `withdrawERC20()` and `withdrawEther()`, add a check via something like `_verifyCallerIsUser()` below.

Note: it is important to be able to specify `controlCodeHash` and `callConfig` in case they have changed.

```
function _verifyCallerIsUser(address EE, bytes32 controlCodeHash, address controller, uint32 callConfig) internal view override {
    if (EE != _getExecutionEnvironmentCustom(msg.sender, controlCodeHash, controller, callConfig)) {
        revert EnvironmentMismatch();
    }
}
```

Fastlane: A potential solution increases gas cost by about 11 000 per metacall, so we have decided to not merge it.

Spearbit: Acknowledged.

5.4.22 Passing of `key` can be simplified

Severity: Gas Optimization

Context: Atlas.sol#L237-L341, Escrow.sol#L46-L227, Escrow.sol#L320-L387, SafetyBits.sol#L69-L116

Description: Some functions (like `_executeSolverOperation()` and `_allocateValue()`) do:

- Pass `key` as a parameter.
- Do `key.hold...Lock()` inside the function.
- Do `key.pack()` inside the function.
- Return `key`.

One function `_getBidAmount()` does:

- Pass `key` as a parameter.
- Do `key.hold...Lock()` inside the function.
- Do `key.pack()` inside the function.
- It doesn't return `key`, but `key` is still updated.

Other functions (like `_executePreOpsCall()`, `_executeUserOperation()`, `_executePostOpsCall()`) do:

- Do `key.hold...Lock()` before the call.
- Pass `key.pack()` as a parameter.

Some functions (like `_bidFindingIteration()`, `_bidKnownIteration()`, `holdPreOpsLock()`, `holdUserLock()`, `holdSolverLock()`, `holdAllocateValueLock()`, `holdPostOpsLock()`) do:

- Pass `key` as a parameter.

- Return key.

The main reasons for the differences are the "*stack too deep*" error. However it would be more consistent and easier to read if the same pattern is used everywhere.

Recommendation: Consider passing the key struct by reference and update it, without having to return it. Note: _getBidAmount() is already doing this. Here is a proof of concept that shows this:

```
// SPDX-License-Identifier: MIT OR Apache-2.0
pragma solidity 0.8.25;
import "hardhat/console.sol";

contract test {
    struct EscrowKey {
        bool solverSuccessful;
    }
    function _executeSolverOperation( EscrowKey memory key) internal pure {
         key.solverSuccessful = true;
    }
    constructor() {
        EscrowKey memory key;
        _executeSolverOperation(key);
        console.logBool(key.solverSuccessful); // true
    }
}
```

Then also do the key.hold...Lock() and key.pack()on the same place. See this issue to further simplify theLock‘ mechanisms:

- Locking mechanism is complicated

Fastlane: Solved in PR 227 and PR 256.

Spearbit: Verified.

5.4.23 Code duplications for call to _allocateValue()

Severity: Gas Optimization

Context: Atlas.sol#L288, Atlas.sol#L328, Atlas.sol#L107-L151

Description: Both _bidFindingIteration() and _bidKnownIteration() are called from execute() and each calls _allocateValue(). It is more logical to do the calls to _allocateValue() from execute() because that is also the place where _executePostOpsCall() is called. This also reduces code duplication and thus reduces deployment size and cost.

```
function execute(/*...*/) /*...*/ {
    // ...
    if (dConfig.callConfig.exPostBids()) {
        (auctionWon, key) = _bidFindingIteration(dConfig, userOp, solverOps, returnData, key);
    } else {
        (auctionWon, key) = _bidKnownIteration(dConfig, userOp, solverOps, returnData, key);
    }
    if (!auctionWon) {
        // ...
     } // else /*...*/ this would be a good place to call _allocateValue()
    // ...
    bool callSuccessful = _executePostOpsCall(auctionWon, returnData, key);
    // ...
}
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    (auctionWon, key) = _executeSolverOperation(/*...*/);
    if (auctionWon) {
        key = _allocateValue(dConfig, solverOps[bidPlaceholder], bidAmounts[bidPlaceholder],
↪   returnData, key);
        key.solverOutcome = uint24(bidPlaceholder);
        return (auctionWon, key);
    }
    // ...
}
function _bidKnownIteration(/*...*/) /*...*/ {
    // ...
    (auctionWon, key) = _executeSolverOperation(/*...*/);
    if (auctionWon) {
        key = _allocateValue(dConfig, solverOp, solverOp.bidAmount, returnData, key);
        key.solverOutcome = uint24(i);
        return (auctionWon, key);
    }
    // ...
}
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.executionSuccessful()) {
        key.solverSuccessful = true;
        return (true, key);   // auctionWon = true
    }
    // ...
}
```

Also see:

- Locking mechanism is complicated

Recommendation: Consider moving the calls to `_allocateValue()` to function `execute()`. For example in the following way:

```
  function execute(/*...*/) /*...*/ {
      // ...
      if (!auctionWon) {
          // ...
      }
+     else {
+         key = _allocateValue(/*...*/);
+     }
      // ...
  }
```

Fastlane: Solved as a result of refactoring.

Spearbit: Verified.

5.4.24 Use `_deposits` instead of the storage variable

Severity: Gas Optimization

Context: GasAccounting.sol#L32-L35

Description: `deposits` is a storage variable and its value is already copied in stack at `_deposits`. `deposits` is still used to read the value.

Recommendation: Update the code as:

```
- fulfilled = deposits >= claims + withdrawals;
+ fulfilled = _deposits >= claims + withdrawals;
```

Fastlane: Solved due to refactoring.

Spearbit: Verified.

5.4.25 `if` conditions always pass

Severity: Gas Optimization

Context: GasAccounting.sol#L94-L115

Description: `reconcile()` first enforces a condition, then does two `if` conditions on the same boolean expression:

```
if (calledBack) revert DoubleReconcile();
// ...
if (/*...*/) {
    if (!calledBack) {
        _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack;
    }
    // ...
}
// ...
if (!calledBack || !fulfilled) {
```

`calledBack` is always false when the execution reaches these `if` conditions, otherwise it'd revert.

Recommendation: Skip checking for `!calledBack`:

```
  if (/*...*/) {
-     if (!calledBack) {
          _solverLock = uint256(uint160(currentSolver)) | _solverCalledBack;
-     }
    // ...
  }
  // ...
- if (!calledBack || !fulfilled) {
+ if (!fulfilled) {
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.26 keccak can be computed at compile time

Severity: Gas Optimization

Context: AtlETH.sol#L149-L151

Description: Following hash is computed at every `permit()` call:

```
keccak256(
    "Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
),
```

This hash can be calculated once and reused since this value is known at compile time.

Recommendation: Compute and store this hash in a constant variable. Refer to OpenZeppelin as an example.

Fastlane: Solved due to refactoring.

Spearbit: Verified.

5.4.27 Structs can be kept in storage

Severity: Gas Optimization

Context: AtlETH.sol#L314-L318, AtlETH.sol#L224-L234. AtlETH.sol#L339-L347

Description: Highlighted code above follow a similar pattern. They copy a storage struct to memory, make changes to it, and then use the copy to update back the storage struct. One example:

```
EscrowAccountAccessData memory aData = accessData[owner];

aData.bonded -= amt;
aData.lastAccessedBlock = uint32(block.number);
accessData[owner] = aData;
```

This has extra gas overhead as the entire struct is copied from and to storage just to update a few struct members. Keeping the struct in storage avoids this overhead.

Recommendation: Update all the highlighted code in the following pattern:

```
- EscrowAccountAccessData memory aData = accessData[owner];
+ EscrowAccountAccessData storage aData = accessData[owner];

  aData.bonded -= amt;
  aData.lastAccessedBlock = uint32(block.number);
- accessData[owner] = aData;
```

This keeps the struct in storage by just creating a reference in `aData`. It makes any changes directly to storage. Thus, there is no need to copy the struct back to storage.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.28 Replace `pendingSurchargeRecipient` with `msg.sender`

Severity: Gas Optimization

Context: AtlETH.sol#L387-L393

Description: `pendingSurchargeRecipient` is read multiple times. Its usage can be replaced with `msg.sender` after it's checked they are equal.

Recommendation:

```diff
  if (msg.sender != pendingSurchargeRecipient) {
      revert InvalidAccess();
  }

- surchargeRecipient = pendingSurchargeRecipient;
+ surchargeRecipient = msg.sender;
  pendingSurchargeRecipient = address(0);
- emit SurchargeRecipientTransferred(surchargeRecipient);
+ emit SurchargeRecipientTransferred(msg.sender);
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.4.29 `SafeMath` can be skipped for Solidity 0.8

Severity: Gas Optimization

Context: SwapMath.sol#L7

Description: Solidity 0.8 has checked math, so `SafeMath` isn't necessary.

Recommendation: Remove `SafeMath` and use vanilla arithmetic operators.

Fastlane: Solved in PR 232.

Spearbit: Verified.

5.4.30 Simulation code path can be kept off-chain

Severity: Gas Optimization

Context: Atlas.sol#L52

Description: Removing simulation mode from contracts simplifies the protocol (removing a lot of branching) and also reduces gas usage.

Recommendation: Off-chain actors can use another version of the code for simulation. This can be done by forking the chain, `vm.etch`ing the simulation code on atlas address and then simulating the execution.

Atlas will provide some off-chain code, the simulation code can be bundled with that. Otherwise, you can just deploy it on a separate address so that interested actors can get the code for simulation trustlessly.

Note: ERC4337 recently did something similar in their v0.7 release (mentioned in the first point here). Their approach is to have an EntryPointSimulations contract that inherits the main contract. The comments say this contract "should never be deployed on-chain and is only used as a parameter for the "eth_call" request".

Fastlane: Acknowledged. For this version of Atlas we will keep the Simulation logic inside the canonical Atlas contract. The main reason for this choice is to keep it simple for parties to simulate `metacalls`, and the `vm.etch` process, while an interesting idea for saving gas, may add a bit of technical friction to the process for external parties.

Spearbit: Acknowledged.

5.4.31 `callSequenceHash` can be gas-optimized

Severity: Gas Optimization

Context: CallVerification.sol#L31-L59

Description: The way `callSequenceHash` is calculated can be changed to reduce its gas consumption. The off-chain software to calculate this hash can be updated too, to match the new onchain version.

Recommendation: Consider these changes:

- Any pre-image data that doesn't contribute to hash uniqueness can be removed: So all indices `i` can be removed.
- `abi.encodeWithSelector(IDAppControl.preOpsCall.selector, userOp)` can be removed since `userOp` is included later and the selector value is fixed.
- Multiple keccaks can be removed in favor of first encoding the pre-image data and then hashing it just once.

```
bytes memory callSequence;
if (dConfig.callConfig & 1 << uint32(CallConfigIndex.RequirePreOps) != 0) {
    // Start with preOps call if preOps is needed
    callSequence = abi.encodePacked(dConfig.to);
}

// then user and solver call
callSequence = abi.encodePacked(callSequence, userOp, solverOps);
callSequenceHash = keccak256(callSequence);
```

Fastlane: Solved in PR 193 and PR 356.

Spearbit: Verified.

5.4.32 `preSolverCall()` can revert instead of returning `false`

Severity: Gas Optimization

Context: ExecutionEnvironment.sol#L182-L196, DAppControl.sol#L81

Description: Decoding return data from `preSolverCall()` can be avoided if it reverts instead of returning `false`.

```
bytes memory data = forwardSpecial(
    abi.encodeWithSelector(IDAppControl.preSolverCall.selector, solverOp, returnData),
    ExecutionPhase.PreSolver
);

(success, data) = control.delegatecall(data);

if (!success) {
    revert AtlasErrors.PreSolverFailed();
}

success = abi.decode(data, (bool));
if (!success) {
    revert AtlasErrors.PreSolverFailed();
}
```

Note: `abi.decode()` itself can also `revert` it the supplied `data` is in an incorrect format.

Recommendation: Consider making this change: `preSolverCall()` doesn't return anything. If, currently, it's returning `true`, return without any data. If it's returning `false`, revert.

With this change, the highlighted code becomes:

```
(success, ) = control.delegatecall(data);

if (!success) {
    revert AtlasErrors.PreSolverFailed();
}
```

Fastlane: Solved in PR 225.

Spearbit: Verified.

5.4.33 Cache `withdrawals`

Severity: Gas Optimization

Context: GasAccounting.sol#L131-L132

Description: The highlighted code can be optimized to avoid calculating new withdrawal amount and reading storage variable `withdrawals` twice.

Recommendation: Update the code as follows:

```
- if (address(this).balance < amount + claims + withdrawals) return false;
- withdrawals += amount;
+ uint _withdrawals = withdrawals + amount;
+ if (address(this).balance < claims + _withdrawals) return false;
+ withdrawals = _withdrawals;
```

Note: after fixing this issue it might not be relevant anymore: "Check with `withdrawals` in `_borrow()` not correct ".

Fastlane: Solved due to refactoring.

Spearbit: Verified.

5.4.34 `metacall()` sends `msg.value` without need

Severity: Gas Optimization

Context: Atlas.sol#L68, SafetyLocks.sol#L46

Description: The function `metacall()` calls `this.execute()` and sends `msg.value`. Because `execute()` is in the same contract this keeps the same amount of `ETH` in the contract. The administration `msg.value` is done via `_setAtlasLock()` and `deposits`, so that is no reason to send `msg.value`.

Note: It would be relevant to send `msg.value` if `execute()` was located in another contract. In that case it might be more logical to send `userOp.value` than `msg.value`.

```
function metacall(/*...*/) /*...*/ {
    // ...
    _setAtlasLock(executionEnvironment, gasMarker, userOp.value);
    try this.execute{ value: msg.value }(/*...*/)
    // ...

function _setAtlasLock(address executionEnvironment, uint256 gasMarker, uint256 userOpValue) internal {
    // ...
    deposits = msg.value;
}
```

Recommendation: Consider removing the sending of `msg.value`.

```
- try this.execute{ value: msg.value }(/*...*/)
+ try this.execute(/*...*/)
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5 Informational

5.5.1 Consider documenting why specific `AtlETH` balances types are used in `_assign()`

Severity: Informational

Context: GasAccounting.sol#L136-L137

Description: In the `_assign()` function, the solver has their `AtlETH` balance reduced. Specifically their `.bonded` balance of `AtlETH` is reduced first, and if that alone is insufficient, their `.unbonding` balance of `AtlETH` is reduced as well. If both amounts combined are still insufficient, the function will not draw from the solver's regular `.balance` of `AtlETH`, and instead will return `isDeficit == true`.

The reason for this behavior is likely that a solver's regular `.balance` of `AtlETH` is meant to be independent and not used within an `Atlas` transaction. In this case, it may be worth explicitly documenting this so the code can be easily understood.

Recommendation: Consider documenting the reasoning for how each `AtlETH` balance type is used in `_assign()`.

Fastlane: Solved in PR 226.

Spearbit: Verified.

5.5.2 `callIndex` and `callCount` don't track the `allocateValue()` call

Severity: Informational

Context: SafetyBits.sol#L96-L107, SafetyLocks.sol#L78

Description: The `callIndex` variable tracks the number of calls that have been executed (or skipped, depending on the `callConfig`) during an Atlas transaction. The `callCount` variable tracks the total number of calls that may be executed during the transaction. However, neither variable tracks the call to the `allocateValue()` function. It may be desirable to include this in the count, as certain `DAppControl` contracts may find this useful.

Recommendation: Consider accounting for the `allocateValue()` call in the `callIndex` and `callCount`. This can be accomplished by adding to the `callCount` initialization:

```
- callCount: solverOpCount + 3,
+ callCount: solverOpCount + 4,
```

and by adding a `callIndex` increment in `holdAllocateValueLock()`:

```
  function holdAllocateValueLock(
      EscrowKey memory self,
      address addressPointer
  )
      internal
      pure
      returns (EscrowKey memory)
  {
      self.lockState = _LOCK_PAYMENTS;
      self.addressPointer = addressPointer;
+     unchecked {
+         ++self.callIndex;
+     }
      return self;
  }
```

Fastlane: Solved due to refactoring in PR 201. The `callIndex` and `callCount` are no longer tracked.

Spearbit: Verified.

5.5.3 Loop counter can be declared with `for` statement

Severity: Informational

Context: SwapIntent.sol#L148-L153, Atlas.sol#L317-L319, Sorter.sol#L43, Sorter.sol#L113, Sorter.sol#L149-L156

Description: Above code declares the `for` loop counter as a separate variable:

```
uint256 i;
for(;/*...*/; i++) {
    // ...
}
```

Recommendation: To restrict the scope of loop counters, consider declaring them with the `for` loop statement:

```
for(uint256 i; /*...*/; i++) {
    // ...
}
```

Fastlane: Solved in PR 188.

Spearbit: Verified.

5.5.4 `TODO`s left in the code

Severity: Informational

Context: See each case listed below.

Description: There are `TODO` comments present in the code. These include:

- Atlas.sol#L58:

  ```
    // NOTE: Currently reverting instead of graceful return to help w/ testing. TODO - still
  ↪   reverting?
  ```

- Atlas.sol#L141:

  ```
    // TODO: point key.addressPointer at bundler if all fail.
  ```

- AtlasVerification.sol#L332-L337:

  ```
    // TODO: consider dapp-owned gas escrow.  Enshrined account
    // abstraction may render that redundant at a large scale, but
    // allocating different parts of the tx to different parties
    // will allow for optimized trustlessness. This could lead to
    // users not having to trust the front end at all - a huge
    // improvement over the current experience.
  ```

- AtlasVerification.sol#L545:

  Note that this would be addressed by the issue "Call to `validateUserOp()` won't work".

  ```
    // TODO: not sure if 30k gas limit is accurate
  ```

- Escrow.sol#L290:

  ```
    // NOTE: Turn this into time stamp check for FCFS L2s?
  ```

- Escrow.sol#L379:

  Note that a full `uint256` does appear safe from overflow in the current code.

  ```
  sub(mload(data), 32) // TODO: make sure a full uint256 is safe from overflow
  ```

- Factory.sol#L174:

  Note that this would be solved by the issue "`Mimic` can be optimized".

  ```
  // TODO: unpack the SHL and reorient
  ```

- GasAccounting.sol#L271:

  Note that this would be solved by the issue "Consider penalizing bundlers for unused gas".

  ```
  // TODO: consider penalizing bundler for too much unused gas (to prevent high escrow
  ↪  requirements for solvers)
  ```

- GasAccounting.sol#L284:

  Note that this is related to the issue "Incorrect `SURCHARGE` multiplication".

  ```
  // TODO: make sure this works w/ the surcharge 10%
  ```

- Storage.sol#L80:

  ```
  // TODO remove these when transient storage behaviour is implemented
  ```

- ExecutionBase.sol#L50:

  ```
  // TODO: simplify this into just the bytes
  ```

- ExecutionBase.sol#L74:

  ```
  // TODO: simplify this into just the bytes
  ```

- SwapIntent.sol#L101:

  ```
  // TODO: If user is Selling Eth, convert it to WETH rather than rejecting.
  ```

- SwapIntent.sol#L103:

  ```
  // TODO: Could maintain a balance of "1" of each token to allow the user to save gas over
  ↪  multiple uses
  ```

- SwapIntent.sol#L190-L191:

  ```
  // TODO: Permit69 is currently enabled during solver phase, but there is low conviction that
  ↪  this
  // does not enable an attack vector. Consider enabling to save gas on a transfer?
  ```

- ChainlinkAtlasWrapper.sol#L7:

  ```
  import "forge-std/Test.sol"; //TODO remove
  ```

- SafetyBits.sol#L6-L8:

  ```
  // TODO remove
  //import {TestUtils} from "../../../test/base/TestUtils.sol";
  // import "forge-std/Test.sol";
  ```

- SafetyBits.sol#L111:

```
  self.addressPointer = address(0); // TODO: Point this to bundler (or builder?) if all solvers
↪   fail
```

Recommendation: Consider addressing each TODO, and consider removing the TODO comments that require no action.

Fastlane: Solved in PR 278 and PR 341.

Spearbit: Verified.

5.5.5 Bundlers reimburse themselves if all solvers fail

Severity: Informational

Context: Atlas.sol#L72-L77

Description: If an Atlas transaction interacts with a `DAppControl` that specifies `needsFulfillment() == false`, it's possible that all `solverOps` fail and `auctionWon == false` will be returned from the `execute()` function. If this happens, the bundler will be treated as the winning solver in the `_settle()` function:

```
(uint256 ethPaidToBundler, uint256 netGasSurcharge) = _settle({
    winningSolver: auctionWon ? solverOps[winningSolverIndex].from : msg.sender,
    bundler: msg.sender
});
```

This implies that bundlers can be required to reimburse themselves for gas costs, which can be inefficient if they don't have an existing `AtlETH` balance to temporarily draw from. This also means that a bundler would be paying the surcharge on gas costs, which can introduce a motivation to intentionally revert the entire transaction to avoid this (which can be achieved with a revert during the final `safeTransferETH()` call).

Recommendation: Consider adjusting the behavior of `_settle()` when `auctionWon == false`. Removing the gas reimbursement and surcharge logic in this scenario may be desirable.

Fastlane: Solved in PR 271.

Spearbit: Verified.

5.5.6 Which party should receive the surplus ETH?

Severity: Informational

Context: AtlasVerification.sol#L70-L177, Atlas.sol#L42-L96, GasAccounting.sol#L254-L298, SafetyLocks.sol#L34-L47

Description: `_validCalls()` allows to be more ETH supplied than required (e.g. `msg.value > userOp.value`). The difference is apperent in `withdrawals` and `deposits`. `_settle()` will give this difference to the winning solver or to `msg.sender` (e.g. the bundler). In case of an error all the ETH is send to `msg.sender` (e.g. the bundler).

Perhaps it is more logical to send the difference to `msg.sender` (e.g. the bundler).

```
function _validCalls(/*...*/) /*...*/ {
     // ...
     // Check that the value of the tx is greater than or equal to the value specified
    if (msgValue < userOp.value) {
        return (userOpHash, ValidCallsResult.TxValueLowerThanCallValue);
    }
    // ...
}
function metacall(/*...*/) /*...*/ {
    // ...
    _setAtlasLock(executionEnvironment, gasMarker, userOp.value);
    try this.execute{ value: msg.value }(dConfig, userOp, solverOps, executionEnvironment, msg.sender,
↪   userOpHash)
```

```
    returns (bool _auctionWon, uint256 winningSolverIndex) {
        (uint256 ethPaidToBundler, /*...*/ ) = _settle({ /*...*/ , bundler: msg.sender });
    } catch (bytes memory revertData) {
        ...
        // Refund the msg.value to sender if it errored
        if (msg.value != 0) SafeTransferLib.safeTransferETH(msg.sender, msg.value);
    }
}
function _settle(/*...*/) /*...*/ {
    // ...
    if (_deposits < _claims + _withdrawals) {
        // ...
        if (_assign(winningSolver, amountOwed, true, false)) {
            revert /*...*/;
        }
    } else {
        // ...
        _credit(winningSolver, amountCredited);
    }
    // ...
    SafeTransferLib.safeTransferETH(bundler, _claims);
    // ...
}
function _setAtlasLock(address executionEnvironment, uint256 gasMarker, uint256 userOpValue) internal {
    // ...
    withdrawals = userOpValue;
    deposits = msg.value;
}
```

Recommendation: Double check which party should receive the surplus ETH. Another approach would be to enforce `msgValue == userOp.value` in `_validCalls()`.

Fastlane: Solved in PR 271. Surplus ETH is calculated in `_settle()` as `_deposits - _withdrawals - netAtlasGasSurcharge - (_claims - _writeoffs)`. This goes to the winning solver and can be seen as subsidizing the solver's gas repayment. If there is no winning solver, surplus ETH goes to the bundler.

Spearbit: Verified.

5.5.7 No error code for failed `_trySolverLock()`

Severity: Informational

Context: Escrow.sol#L97-L168, GasAccounting.sol#L225-L246, EscrowBits.sol#L66-L70

Description: If `_trySolverLock(solverOp) == false` there is no error assigned to `result`, it just stays `0`. The `0` value is later on used in the `updateEscrow()` check and in an `emit`. The `updateEscrow()` check is less fine grained this way. Troubleshooing on basis of the emitted event is somewhat difficult.

```
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.canExecute() && _trySolverLock(solverOp)) {
        // ...
    }
    // ...
    key.solverOutcome = uint24(result); // result could be 0 if _trySolverLock fails
    _releaseSolverLock(/*...*/, result, /*...*/);
    // ...
    emit SolverTxResult(solverOp.solver, solverOp.from, result.executedWithError(), false, result); //
↪   result could be 0
    // ...
}
function _releaseSolverLock(/*...*/) /*...*/ {
    // ...
    if (!bidFind && !result.updateEscrow()) return; // result could be 0
    // ...
}
function updateEscrow(uint256 result) internal pure returns (bool) {
    return (result & _NO_REFUND == 0);
}
```

Recommendation: Consider assigning an error code to `result` in case `_trySolverLock(solverOp)` fails.

Fastlane: Solved in PR 225 as a failing `_trySolverLock()/_borrow()` now reverts with an `InsufficientEscrow` error which is caught and reflected in `result`.

Spearbit: Verified.

5.5.8 Parameter `callConfig` seems redundant

Severity: Informational

Context: Permit69.sol#L56-L107, Atlas.sol#L54, Atlas.sol#L381-L385, Factory.sol#L64-L71, Factory.sol#L125-L151, ExecutionEnvironment.sol#L350-L359, ExecutionEnvironment.sol#L383-L392

Description: In `transferUserERC20()` and `transferDAppERC20()`, which call `_verifyCallerIsExecutionEnv()` a check is done for `msg.sender == _getExecutionEnvironmentCustom`. This can only be the case with the current `callConfig`, because `metacall()` / `_getOrCreateExecutionEnvironment()` retrieves `getDAppConfig(userOp).callConfig`.

However `transferUserERC20()` and `transferDAppERC20()` allow specifing `callConfig`, which only has added value if `getDAppConfig(userOp).callConfig` changes in between calls in the same transaction, which seems far fetched.

```
function transferUserERC20(..., address user, address controller, uint32 callConfig ,/*...*/) /*...*/ {
    _verifyCallerIsExecutionEnv(user, controller, callConfig);
}
function transferDAppERC20(..., address user, address controller, uint32 callConfig ,/*...*/) /*...*/ {
    _verifyCallerIsExecutionEnv(user, controller, callConfig);
}
function _verifyCallerIsExecutionEnv(address user, address controller, uint32 callConfig) internal view
↪   override {
    if (msg.sender != _getExecutionEnvironmentCustom(user, controller.codehash, controller,
↪   callConfig)) {
        revert EnvironmentMismatch();
    }
}
/// @notice Generates the address of a user's execution environment affected by deprecated callConfig
↪   changes in the
/// DAppControl.
function _getExecutionEnvironmentCustom(/*...*/) /*...*/ {
    // generate address based on user, controlCodeHash, controller, callConfig, _salt
}
```

```
function metacall(/*...*/) /*...*/ {
    /*...*/
    (address executionEnvironment, /*...*/ ) = _getOrCreateExecutionEnvironment(userOp);
    /*...*/
}
function _getOrCreateExecutionEnvironment(/*...*/) /*...*/ {
    address control = userOp.control;
    dConfig = IDAppControl(control).getDAppConfig(userOp);
    executionEnvironment = _setExecutionEnvironment(control, userOp.from, dConfig.callConfig,
↪   control.codehash);
}
```

Recommendation: Check the usefulness of the parameter `callConfig` in `transferUserERC20()` and `transferDAppERC20()` and consider removing it.

If `callConfig` does change over time, it might be useful for an `ExecutionEnvironment` to retrieve any left over funds in a `sister ExecutionEnvironment` which is based on a previous `callConfig`. If that is relevant then `withdrawERC20()` and `withdrawEther()` could be adapted to allow that.

Fastlane: Solved in PR 270.

Spearbit: Verified.

5.5.9 Prevent abuse of `transferDAppERC20()`

Severity: Informational

Context: Permit69.sol#L88-L107

Description: If a `userOp` does a `delegatecall`, the destination contract could use `transferDAppERC20()` to transfer tokens from the `DappControl` contract. This is a feature but could also be abused.

Recommendation: If a `dappControl` allows `delegatecall` and has allowances to `Atlas`, it should check / enforce limits on the destination that is called by the `userOp`.

Fastlane: Solved in PR 227. The `delegatecalled` contract from the `userOp` can't access `transferDAppERC20()` anymore, as the the user operation phase is not part of the `SAFE_DAPP_TRANSFER` phase set.

Spearbit: Verified.

5.5.10 How to prevent malicious `DappControl` contracts

Severity: Informational

Context: DAppControl.sol

Description: A malicious `DappControl` contract could move all of the user's tokens, for which they have given an allowance to `Atlas`. This can be done via `transferUserERC20()`.

A malicious `DappControl` contract could also set allowances for random tokens of an `ExecutionEnvironment`. That might be abused later on in combination with an external call from the `ExecutionEnvironment`.

Recommendation: There should be a mechanism to make sure a user interacts with a valid `DappControl`. This could be off-chain in a user interface. Or it could be on-chain via an allowlist, possibly using the Ethereum Attestation Service.

Another approach could be to limit the types and amounts of tokens that could be transferred in the `UserOp`.

***Note:** a user could also limit this by limiting their allowances to `Atlas`.*

Fastlane: The frontend should be responsible for only letting users interact with audited and off-chain whitelisted (by the frontend) `DappControl` contracts. Won't fix.

Spearbit: Acknowledged.

5.5.11 `_availableFundsERC20()` is optional which is not clear

Severity: Informational

Context: ExecutionBase.sol#L226-L263

Description: The function `_availableFundsERC20()` does several checks to make sure `transferUserERC20()` and/or `transferDAppERC20()` will succeed. Functions `transferUserERC20()` and/or `transferDAppERC20()` check these limits again so the use of `_availableFundsERC20()` is optional, but this might not be clear.

```
function _availableFundsERC20(/*...*/) /*...*/ {
    // checks balance, phase, source and allowance
}
```

Recommendation: Consider adding a comment about the usage of `_availableFundsERC20()`.

Fastlane: Solved in PR 219.

Spearbit: Verified.

5.5.12 Internal functions `forward()` and `forwardSpecial()` not prefixt with _

Severity: Informational

Context: ExecutionBase.sol#L49-L76

Description: `forward()` and `forwardSpecial()` functions are internal but the function name doesn't start with `_`, unlike most other internal functions.

```
function forward(bytes memory data) internal pure returns (bytes memory) {
    // ...
}
function forwardSpecial(bytes memory data, ExecutionPhase phase) internal pure returns (bytes memory) {
    // ...
}
```

Recommendation: Consider prefixing the functions with `_`.

Fastlane: Solved in PR 169.

Spearbit: Verified.

5.5.13 Outdated comment in `_allocateValueCall()` of `V2DAppControl`

Severity: Informational

Context: V2DAppControl.sol#L117-L151

Description: `_allocateValueCall()` has an outdated comment because governance tokens are not burnt but sent to the user.

```
function _allocateValueCall(address, uint256 bidAmount, bytes calldata) internal override {
    // ...
    /*
    console.log("Governance Tokens Burned:", govIsTok0 ? amount0Out : amount1Out);
    */
    // ...
}
```

Recommendation: Consider updating the text.

Fastlane: Solved in PR 203.

Spearbit: Verified.

5.5.14 `transfer()` is used

Severity: Informational

Context: V2DAppControl.sol#L126

Description: `V2DAppControl` uses `transfer()` which isn't best practice.

```
function _allocateValueCall(address, uint256 bidAmount, bytes calldata) internal override {
    // ...
    ERC20(WETH).transfer(WETH_X_GOVERNANCE_POOL, bidAmount);
    // ...
}
```

Recommendation: Consider using `safeTransfer()`.

Fastlane: Solved in PR 213.

Spearbit: Verified.

5.5.15 Math calculations could be bundled

Severity: Informational

Context: V2DAppControl.sol#L131-L137, SwapMath.sol

Description: Function `_allocateValueCall()` contains some math calculations and the library `SwapMath` also contains math calculations. For consistency they could be bundled in one place.

```
function _allocateValueCall(address, uint256 bidAmount, bytes calldata) internal override {
    // ...
    if (govIsTok0) {
        amount0Out = ((997_000 * bidAmount) * uint256(token0Balance))
            / ((uint256(token1Balance) * 1_000_000) + (997_000 * bidAmount));
    } else {
        amount1Out = ((997_000 * bidAmount) * uint256(token1Balance))
            / (((uint256(token0Balance) * 1_000_000) + (997_000 * bidAmount)));
    }
    // ...
}
```

Recommendation: Consider moving the math calculations to library `SwapMath`.

Fastlane: Solved in PR 202.

Spearbit: Verified.

5.5.16 Array lengths not checked in `transmit()` nor `_verifyTransmitData()`

Severity: Informational

Context: ChainlinkAtlasWrapper.sol#L77-L124, ChainlinkDAppControl.sol#L136-L162

Description: Neither `transmit()` nor `_verifyTransmitData()` nor `verifyTransmitSigners()` checks the lenght of `rs` versus `ss`.

In `_verifyTransmitData()`:

- When `r.observations.length == 0` then the for loop will revert.
- `r.observations.length` might be able to be compared to `rs.length`.

In `verifyTransmitSigners()`:

- If `rs.length > MAX_NUM_ORACLES` then `rawVs[i]` will get out of bounds because it is only 32 bytes large.

These situations could be used to detect mismatches and revert with a clear message.

```
function transmit(/*...*/, bytes32[] calldata rs, bytes32[] calldata ss, /*...*/) /*...*/ {
    // ...
    int256 answer = _verifyTransmitData(report, rs, ss, rawVs);
    // ...
}
function _verifyTransmitData(/*...*/, bytes32[] calldata rs, bytes32[] calldata ss, /*...*/) /*...*/ {
    // ...
    for (uint256 i = 0; i < r.observations.length - 1; ++i) { // revert when length == 0
        // ...
    }
    bool signersVerified =
            IChainlinkDAppControl(DAPP_CONTROL).verifyTransmitSigners(address(BASE_FEED), report, rs,
↪   ss, rawVs);
}
function verifyTransmitSigners(/*...*/, bytes32[] calldata rs, bytes32[] calldata ss,/*...*/) /*...*/ {
    // ...
    for (uint256 i = 0; i < rs.length; ++i) {
        /*...*/ uint8(rawVs[i]) /*...*/  // could get out of bounds
    }
    // ...
}
```

Recommendation: Consider checking `rs.length == ss.length`. Compare `r.observations.length` to `rs.length`. Check for `r.observations.length==0`. Check `rs.length <= MAX_NUM_ORACLES` Note: as the three functions are related the following checks don't have to be present in all functions.

Fastlane: Solved in PR 205.

Spearbit: Verified.

5.5.17 Old term metaFlashCall used

Severity: Informational

Context: OEV.t.sol#L640-L643, SwapIntent.t.sol#L348-L351, OEValt.t.sol#L631-L634, Accounting.t.sol#L246-L249

Description: Several test files mention metaFlashCall. This is a deprecated term.

```
// This ensures a function can only be called through metaFlashCall
// which includes security checks to work safely with Atlas
modifier onlySelf() {
    require(msg.sender == address(this), "Not called via metaFlashCall");
    _;
}
```

Recommendation: Consider replacing the term metaFlashCall.

Fastlane: Solved in PR 220.

Spearbit: Verified.

5.5.18 Sorting same bids result in reverse order

Severity: Informational

Context: Sorter.sol#L133-L175

Description: If multiple Solvers have the same bid then the sorted order is the reverse of the input order. This is due to the >= operator in the if statement.

```
function _sort(/*...*/) /*...*/ {
    // ...
    for (j = 0; j < count;) {
        if (sortingData[j].valid && sortingData[j].amount >= topBidAmount) {
            topBidAmount = sortingData[j].amount;
            topBidIndex = j;
        }
        unchecked { ++j; }
    }
    // ...
}
```

Recommendation: If reversing the order is unwanted, consider refactoring the code. If > is used then it would be the same order. However > has an edge case if all amounts are 0, then topBidIndex would always be 0 and thus sorted[i] too. It could be solved by changing topBidAmount and topBidIndex to int256, and initialize them to -1.

This assumes the topBidAmount is lower than type(int256).max.

Fastlane: Solved in PR 207.

Spearbit: Verified.

5.5.19 Initialization with `0` is inconsistent

Severity: Informational

Context: Sorter.sol#L43, DAppIntegration.sol#L153, AtlasVerification.sol#L687-L697

Description: Sometimes variables are initialized with 0, and sometimes they are not initialized. This is not consistent.

```
uint256 i = 0;
```

```
for (uint256 i = 0; i < /*...*/ ; i++) {
    // ...
}
```

Recommendation: Consider using the same pattern everywhere, for example not initialized variables that should stay at `0`.

Fastlane: Solved in PR 221.

Spearbit: Verified.

5.5.20 `unchecked` not necessary in `for` loops

Severity: Informational

Context: Atlas.sol#L306-L338, Sorter.sol#L45-L50, Sorter.sol#L115-L127, Sorter.sol#L152-L171, CallVerification.sol#L53-L64, SwapIntent.sol#L153-L162

Description: `for` loop index parameter increments no longer need `unchecked` in Solidity 0.8.22, see Solidity docs unchecked-loop-increment.

```
function _bidKnownIteration(/*...*/) /*...*/ {
    // ...
    for (; i < k;) {
        // ...
        unchecked { ++i; }
    }
}
```

Recommendation: Consider removing `unchecked` from for loop index parameters.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.21 Double negations in comments

Severity: Informational

Context: EscrowBits.sol#L66-L70

Description: The comments in function `updateEscrow()` use double negations, which are difficult to read.

```
function updateEscrow(uint256 result) internal pure returns (bool) {
    // dont update solver escrow if they don't need to refund gas
    // returns true is solver doesn't get to bypass the refund.
    return (result & _NO_REFUND == 0);
}
```

Recommendation: Consider changing the comments to:

```
  function updateEscrow(uint256 result) internal pure returns (bool) {
-     // dont update solver escrow if they don't need to refund gas
-     // returns true is solver doesn't get to bypass the refund.
+     // Only update solver escrow if they need to refund gas
+     // returns true if solver has to do the refund.
      return (result & _NO_REFUND == 0);
  }
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.22 Function `getCallChainHash()` could use `needsPreOpsCall()`

Severity: Informational

Context: CallBits.sol#L111-L113, CallVerification.sol#L19-L65

Description: Function `getCallChainHash()` does a check with `RequirePreOps`. It could also use the function `needsPreOpsCall()` for easier maintainability and readability.

```
function getCallChainHash(/*...*/) /*...*/ {
    // ...
    if (dConfig.callConfig & 1 << uint32(CallConfigIndex.RequirePreOps) != 0) {
        // ...
    }
    // ...
}
function needsPreOpsCall(uint32 callConfig) internal pure returns (bool needsPreOps) {
    needsPreOps = (callConfig & 1 << uint32(CallConfigIndex.RequirePreOps) != 0);
}
```

Recommendation: Consider using `needsPreOpsCall()` in `getCallChainHash()`.

Fastlane: Solved in PR 186.

Spearbit: Verified.

5.5.23 `modifier payBids()` can simplified

Severity: Informational

Context: SolverBase.sol#L29-L91

Description: `modifier payBids()` retrieves ETH for both the bid and the return of `msg.value`. This is not straightforward:

```
modifier payBids(address bidToken, uint256 bidAmount) {
    // ...
    _;
    if (bidToken == address(0)) { // Ether balance
        uint256 ethOwed = bidAmount + msg.value;
        if (ethOwed > address(this).balance) {
            IWETH9(WETH_ADDRESS).withdraw(ethOwed - address(this).balance);
        }
        SafeTransferLib.safeTransferETH(msg.sender, bidAmount);
    } else { // ERC20 balance
        if (msg.value > address(this).balance) {
            IWETH9(WETH_ADDRESS).withdraw(msg.value - address(this).balance);
        }
        SafeTransferLib.safeTransfer(ERC20(bidToken), msg.sender, bidAmount);
    }
}
```

Recommendation: Consider simplifying the code in the following way. This is less efficient but easier to understand, as this is just an example.

```
modifier safetyFirst(address sender) {
    // ...
    _;
    // ...
    if (msg.value > address(this).balance) {
        IWETH9(WETH_ADDRESS).withdraw(msg.value - address(this).balance);
    }
    IEscrow(_atlas).reconcile{ value: msg.value }(msg.sender, sender, shortfall);
}
modifier payBids(address bidToken, uint256 bidAmount) {
    // ...
    _;
    if (bidToken == address(0)) { // Ether balance
        if (bidAmount > address(this).balance) {
            IWETH9(WETH_ADDRESS).withdraw(bidAmount - address(this).balance);
        }
        SafeTransferLib.safeTransferETH(msg.sender, bidAmount);
    } else { // ERC20 balance
        SafeTransferLib.safeTransfer(ERC20(bidToken), msg.sender, bidAmount);
    }
}
```

Fastlane: Solved in PR 239 and PR 335.

Spearbit: Verified.

5.5.24 Solver knows error codes of previous solvers

Severity: Informational

Context: SafetyBits.sol#L52-L67, Escrow.sol#L97-L168, Escrow.sol#L423-L474

Description: `_executeSolverOperation()` is executed for multiple `solvers`. The `result` is stored in `key.solverOutcome` and this value is passed to the next `solver`. It is unclear why the next `solver` should want to know the error code of the previous `solver`. Perhaps that information can somehow be abused. We would expect the `solvers` to be isolated.

```
function _executeSolverOperation(
    // ...
    result |= _solverOpWrapper(/*...*/ , key.pack());
    key.solverOutcome = uint24(result);
    if (result.executionSuccessful()) {
        // ...
        return (true, key);
    }
    // ...
}
function _solverOpWrapper(/*...*/, bytes32 lockBytes) /*...*/ {
    // ...
    bytes memory data = abi.encodeWithSelector(... solverMetaTryCatch.selector, /*...*/);
    data = abi.encodePacked(data, lockBytes);
    (success, data) = environment.call{ value: solverOp.value }(data);
}
function pack(EscrowKey memory self) internal pure returns (bytes32 packedKey) {
    packedKey = bytes32(
        abi.encodePacked(
            // ...
            self.solverOutcome,
            // ...
        )
    );
}
```

Recommendation: Consider not passing `self.solverOutcome` to the solvers.

Fastlane: Solved in PR 225 as `Atlas` calls the `solver` contract directly now, and `lockBytes` is not appended as calldata on this call.

Spearbit: Verified.

5.5.25 Two different ways to represent errors

Severity: Informational

Context: ValidCallsTypes.sol#L4-L24, EscrowTypes.sol#L18-L43

Description: There are two approaches to store error information. One by storing one error in a `uint256`, via `ValidCallsResult`. The other by storing multiple errors as separate bits in a `uint256`, via `SolverOutcome`. The errors also partly overlap. This approach could be confusing and error prone.

```
enum ValidCallsResult {
    Valid,
    GasPriceHigherThanMax,
    TxValueLowerThanCallValue,
    DAppSignatureInvalid,
    UserSignatureInvalid,
    TooManySolverOps,
    UserDeadlineReached,
    DAppDeadlineReached,
    ExecutionEnvEmpty,
    NoSolverOp,
    UnknownAuctioneerNotAllowed,
    InvalidSequence,
    InvalidAuctioneer,
    InvalidBundler,
    OpHashMismatch,
    DeadlineMismatch,
    InvalidControl,
    InvalidSolverGasLimit,
    InvalidDAppNonce
}
enum SolverOutcome {
    // no refund (relay error or hostile user)
    InvalidSignature,
    InvalidUserHash,
    DeadlinePassedAlt,
    InvalidTo,
    UserOutOfGas,
    AlteredControl,
    // Partial Refund but no execution
    DeadlinePassed,
    GasPriceOverCap,
    InvalidSolver,
    PerBlockLimit, // solvers can only send one tx per block
    // if they sent two we wouldn't be able to flag builder censorship
    InsufficientEscrow,
    GasPriceBelowUsers,
    CallValueTooHigh,
    // execution, with full user refund
    PreSolverFailed,
    SolverOpReverted,
    PostSolverFailed,
    IntentUnfulfilled,
    BidNotPaid,
    BalanceNotReconciled,
    EVMError
}
```

Recommendation: Consider harmonizing the approaches.

Fastlane: Added some comments in PR 269 but kept the current logic.

Spearbit: Acknowledged.

5.5.26 Structs with limited comments

Severity: Informational

Context: DAppApprovalTypes.sol#L8-L27, SolverCallTypes.sol#L8-L23, UserCallTypes.sol#L8-L21

Description: Several `structs` have elements without comments.

Recommendation: Consider adding more comments, here are a few suggestions:

```
  struct UserOperation {
      // ...
+     uint256 value; // Amount of ETH required by the UserOperation
      uint256 gas;
      uint256 maxFeePerGas;
+     uint256 nonce; // >0, < type(uint128).max
+     uint256 deadline; // can be 0, based on block.number
      // ...
+     address sessionKey; // temporary key autorized to sign the DAppOperation
      bytes data;
      bytes signature;
  }
  struct DAppOperation {
      // ...
      uint256 value;
      uint256 gas;
      uint256 nonce;  // >0, < type(uint128).max
      uint256 deadline; // can be 0, based on block.number
      // ...
      bytes signature;
  }
  struct DAppConfig {
+     address to; // DappControl address
      uint32 callConfig;
+     address bidToken;  // 0 for ETH
      uint32 solverGasLimit;
  }
  struct SolverOperation {
      // ...
      uint256 value;
      uint256 gas;
      // ...
+     uint256 deadline; // // can be 0(?), based on block.number
     ...
+     address bidToken;  // 0 for ETH
      uint256 bidAmount;
      bytes data;
      bytes signature;
  }
```

Fastlane: Solved in PR 254.

Spearbit: Verified.

5.5.27 Failed paymentsSuccessful might go undetected

Severity: Informational

Context: Escrow.sol#L183-L206

Description: In _allocateValue(), if the call to allocateValue() fails then key.paymentsSuccessful is kept at false, and processing continues. This value is passed to the PostOps Hook, so that hook could take action on it. However this is not used in any of the examples.

The function metacall() doesn't return any information about this, so if this situation occurs it might be difficult to detect for the caller. Also this value isn't emited anywhere, so it is also difficult to track offline.

```
function _allocateValue(/*...*/) /*...*/ {
    // ...
    bytes memory data = abi.encodeWithSelector(IExecutionEnvironment.allocateValue.selector, ... ));
    // ...
    (bool success,) = key.executionEnvironment.call(data);
    if (success) {
        key.paymentsSuccessful = true;
    }
    return key;
}
```

Recommendation: Consider doing one or more of the following:

- Adding more comments for the PostOps Hook.
- Returning the paymentsSuccessful value from metacall().
- Doing an emit of the paymentsSuccessful value.

Fastlane: Solved in PR 247. Added a config option allowing allocateValue() to fail silently, and a comment that points out that paymentsSuccessful value can be accessed by the dApp control during the postOps hook.

Spearbit: Verified.

5.5.28 No minimum value for `ESCROW_DURATION`

Severity: Informational

Context: Storage.sol#L61-L89

Description: The constructor of `Storage` doesn't enforce any limits on `_escrowDuration`. A too short duration might accidentally be set which will allow unbonding of `AtlETH` in an unexpected short period.

```
constructor(uint256 _escrowDuration, /*...*/ ) /*...*/ {
    ESCROW_DURATION = _escrowDuration;
    // ...
}
```

Recommendation: Consider enforcing a minimum duration for `_escrowDuration`.

Fastlane: Solved in PR 262, which ensures ESCROW_DURATION is not zero. It's hard to decide on a minimum here as it changes a lot depending on block rate.

Spearbit: Verified.

5.5.29 Consider penalizing bundlers for unused gas

Severity: Informational

Context: GasAccounting.sol#L271

Description: For a solver to be considered successful in an Atlas transaction, they must pass a `validateBalances()` check at the end of the `solverMetaTryCatch()` function. This check will ensure that the Atlas `deposits` are larger than the Atlas `claims + withdrawals`, where `claims` is an upper-bound gas cost based on the starting `gasleft()`. Since a bundler can provide excess gas that will later be refunded to them, the `claims` value may be inflated at this point. While this unused gas is later subtracted in the `_settle()` function and the winning solver is reimbursed the excess ETH, this behavior implies an inefficiency. With a high initial `gasleft()` value, the winning solver would be required to provide otherwise unnecessary ETH which will be immediately returned to them.

Recommendation: Consider applying a small penalty to bundlers who use a large amount of unused gas. Note that this appears to already be considered for future work, according to the following comment:

```
// TODO: consider penalizing bundler for too much unused gas (to prevent high escrow requirements for
↪  solvers)
```

Fastlane: Addressed in PR 271.

In `_adjustAccountingForFees()` we calculate an upper estimate of what we expect the `gasleft()` at this point of the metacall should be, given the solver count and index of the winning solver. `writeoffs` is then increased in proportion to how much the actual `gasleft()` exceeds our estimated upper bound. This increase in `writeoffs` ultimately reduces the gas refund sent to the bundler.

Spearbit: Verified. After an internal discussion with the Fastlane team, it was determined that this solution does not remove all potential ways for a bundler to achieve a high escrow requirement for solvers, but it partially solves the problem and this tradeoff is better than further complicating the code.

5.5.30 Same nonce storage used for `userOp.from` and `dAppOp.from`

Severity: Informational

Context: AtlasVerification.sol#L386

Description: The `_handleNonces()` function is used by both the `_verifyUser()` function (to invalidate the `userOp.from` nonce) and the `_verifyDApp()` function (to invalidate the `dAppOp.from nonce`). Regardless of the scenario, the underlying account address is treated the same in storage. So, if an address is sometimes the `userOp.from` value and other times the `dAppOp.from` value, the `nonce` management can be complicated. The most complex scenario would be if `userOp.from == dAppOp.from` in a single `Atlas` transaction. To allow simpler `nonce` management and better sequencing, using separate storage may be desirable in these edge cases.

Recommendation: Consider differentiating the nonce storage based on the use case of the nonce.

Fastlane: Solved in PR 253.

Spearbit: Verified.

5.5.31 Example code `atlasSolverCall` is limited

Severity: Informational

Context: SolverBase.sol#L29-L91

Description: Function `atlasSolverCall` of `SolverBase` has no code for the `invertsBidValue` case.

Function `atlasSolverCall` also returns exactly the `bidAmount` that was supplied as a parameter to `atlasSolverCall()`. Although it retrieves bidBalance at the beginning of the function via modifier `payBids()`, it doesn't use it.

```
modifier payBids(address bidToken, uint256 bidAmount) {
    // Track starting balances
    uint256 bidBalance =    // not used
        bidToken == address(0) ? address(this).balance - msg.value :
↪   ERC20(bidToken).balanceOf(address(this));
    _;
    // ...
}
```

Recommendation: Consider adding example code for the `invertsBidValue` case. Also consider adding example code for the situation more ERC20 tokens are returned or less ERC20 tokens are retrieved.

The following table summarized to send ETH and ERC20s for the `!invertsBidValue` case, which is currently present in the code:

!invertsBidValue	ETH bidToken	ERC20 bidToken
ETH transfer	msg.value + bidAmount	msg.value
ERC20 transfer	-	send bidAmount

The following table shows the values that could be send with the `invertsBidValue` case:

invertsBidValue	ETH bidToken	ERC20 bidToken
ETH transfer	msg.value - bidAmount	msg.value
ERC20 tranfer	-	retrieve bidAmount

Fastlane: Solved in PR 257.

Spearbit: Verified.

5.5.32 Code duplication in `solverMetaTryCatch()`

Severity: Informational

Context: ExecutionEnvironment.sol#L142-L317

Description: Function `solverMetaTryCatch()` contains some code duplication. For easier code maintenance and contract size reduction these parts could be combined.

```
function solverMetaTryCatch(/*...*/) /*...*/ {
    // ...
    if (_bidFind()) {
        // ...
        if (endBalance > 0) {
            IEscrow(atlas).contribute{ value: endBalance }();
        }
        (, success) = IEscrow(atlas).validateBalances();
        if (!success) revert AtlasErrors.BalanceNotReconciled();
        // ...
        revert AtlasErrors.BidFindSuccessful(netBid);
    }
    // ...
    if (endBalance > 0) {
        IEscrow(atlas).contribute{ value: endBalance }();
    }
    (, success) = IEscrow(atlas).validateBalances();
    if (!success) {
        revert AtlasErrors.BalanceNotReconciled();
    }
}
```

Recommendation: Consider changing the code to:

```
  function solverMetaTryCatch(...) ... {
      // ...
      if (_bidFind()) {
          // ...
-          if (endBalance > 0) {
-              IEscrow(atlas).contribute{ value: endBalance }();
-          }
-          (, success) = IEscrow(atlas).validateBalances();
-          if (!success) revert AtlasErrors.BalanceNotReconciled();
-          // ...
-          revert AtlasErrors.BidFindSuccessful(netBid);
-      }
+     else {
          // ...
      }
      if (endBalance > 0) {
          IEscrow(atlas).contribute{ value: endBalance }();
      }
      (, success) = IEscrow(atlas).validateBalances();
      if (!success) {
          revert AtlasErrors.BalanceNotReconciled();
      }
+     if (_bidFind()) {
+         revert AtlasErrors.BidFindSuccessful(netBid);
  }
```

Fastlane: Refactored in PR 225 when we changed solver called to happen directly from Atlas to solver contract, and the pre and post solver call logic each have their own hooks in the Execution Environment.

Spearbit: Verified.

5.5.33 `withdrawSurcharge()` might be done too early

Severity: Informational

Context: Storage.sol#L61-L89, AtlETH.sol#L355-L364, GasAccounting.sol#L254-L298

Description: The `surcharge` is initialized in the `constructor` of `Storage` and increased via `_settle()`. Then it can be withdrawn via `withdrawSurcharge()`. However if this is done too early on and not enough ETH is present in the `Atlas` contract, then flashloans are not possible. This is not obvious.

```
constructor(/*...*/) payable {
    // ...
    // Initialized with msg.value to seed flash loan liquidity
    surcharge = msg.value;
    // ...
}
function withdrawSurcharge() external {
    if (msg.sender != surchargeRecipient) {
        revert InvalidAccess();
    }
    uint256 paymentAmount = surcharge;
    surcharge = 0; // Clear before transfer to prevent reentrancy
    SafeTransferLib.safeTransferETH(msg.sender, paymentAmount);
    emit SurchargeWithdrawn(msg.sender, paymentAmount);
}
function _settle(/*...*/) /*...*/ {
    // ...
    uint256 _surcharge = surcharge;
    // ...
    surcharge = _surcharge + netGasSurcharge;
    // ...
}
```

Recommendation: Consider adding a comment at function `withdrawSurcharge()`. Consider allowing `withdrawSurcharge()` to withdraw a part of the `surcharge`.

Fastlane: Solved in PR 224 by adding a comment above the function about the risks.

Spearbit: Verified.

5.5.34 Existence of both `SURCHARGE` and `surcharge`

Severity: Informational

Context: Storage.sol#L12-L47

Description: `contract Storage` contains both `SURCHARGE` and `surcharge` which might be confusing.

```
contract Storage is AtlasEvents, AtlasErrors {
    uint256 public constant SURCHARGE = 1_000_000; // Out of 10_000_000
    uint256 public surcharge; // Atlas gas surcharges
}
```

Recommendation: Consider changing the names and adding comments, for example in the following way:

```
- uint256 public constant SURCHARGE = 1_000_000; // Out of 10_000_000
+ uint256 public constant SURCHARGE_RATE = 1_000_000; // surcharge rate Out of 10_000_000
- uint256 public surcharge; // Atlas gas surcharges
+ uint256 public cumulativeSurcharge; // Cumulative gas surcharges collected.
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.35 Direct access to `accessData[]`, `_balanceOf[]` and `bondedTotalSupply`

Severity: Informational

Context: Escrow.sol#L285, GasAccounting.sol#L86, GasAccounting.sol#L145, GasAccounting.sol#L150, GasAccounting.sol#L154-L161, GasAccounting.sol#L182, GasAccounting.sol#L194, GasAccounting.sol#L199

Description: Several contracts outside of `AtlETH` access `accessData[]`, `_balanceOf[]` and `bondedTotalSupply` directly, however it better to hide the implementation details. The updates outside of the `AtlETH` don't have `emit` attached so for an offchain indexer it is difficult to track all `AtlETH` movements.

Recommendation: Concentrate all code that touches `accessData[]` in `AtlETH` and use functions to access them. Also consider adding `emit Bond()` / `emit Unbond()` / `emit Redeem()`.

Fastlane: Won't fix due to gas reasons.

Spearbit: Acknowledged.

5.5.36 Incorrent comment in `reconcile()`

Severity: Informational

Context: GasAccounting.sol#L73-L84

Description: The comment `approvedAmount` should probably be `maxApprovedGasSpend`.

```
function reconcile(...,  uint256 maxApprovedGasSpend) /*...*/ {
    // NOTE: approvedAmount is the amount of the solver's atlETH that the solver is allowing
    // ...
}
```

Recommendation: Consider changing the comment to:

```
- // NOTE: approvedAmount is the amount of the solver's atlETH that the solver is allowing
+ // NOTE: maxApprovedGasSpend is the amount of the solver's atlETH that the solver is allowing
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.37 validateBalances() and _checkAtlasIsUnlocked() could use isUnlocked()

Severity: Informational

Context: GasAccounting.sol#L29-L38, SafetyLocks.sol#L89-L96, SafetyLocks.sol#L105-L107, DAppIntegration.sol#L26, DAppIntegration.sol#L163-L165

Description: validateBalances() checks for a lock via the value of _deposits. It is safer and more readable to check this via the lock itself. _checkAtlasIsUnlocked() checks for a lock via the value UNLOCKED. It is more readable to use isUnlocked().

```
function validateBalances() /*...*/ {
  // ...
    // Check if locked.
    if (_deposits != type(uint256).max) {
        // ...
    }
    // ...
}
function isUnlocked() external view returns (bool) {
    return lock == UNLOCKED;
}
function _releaseAtlasLock() internal {
    // ...
    lock = UNLOCKED;
    // ...
    deposits = type(uint256).max;
}
```

```
address internal constant UNLOCKED = address(1);
function _checkAtlasIsUnlocked() internal view {
    if (IAtlas(ATLAS).lock() != UNLOCKED) revert AtlasErrors.AtlasLockActive();
}
```

Recommendation: For validateBalances() consider changing the code to:

```
  function validateBalances() /*...*/ {
    // ...
      // Check if locked.
-     if (_deposits != type(uint256).max) {
+     if (!isUnlocked()) {
          // ...
      }
      // ...
  }
```

Note: This might no longer be necessary after this issue is solved: "Moving validateBalances() to Atlas".

For _checkAtlasIsUnlocked() consider changing the code to:

```
- address internal constant UNLOCKED = address(1);
  function _checkAtlasIsUnlocked() internal view {
-    if (IAtlas(ATLAS).lock() != UNLOCKED) revert AtlasErrors.AtlasLockActive();
+    if (!ISafetyLocks(ATLAS).isUnlocked()) revert AtlasErrors.AtlasLockActive();
  }
```

Fastlane: Solved in PR 225.

Spearbit: Verified.

5.5.38 Future authorization might fail because solver contract isn't `solverOp.from`

Severity: Informational

Context: SolverBase.sol#L29-L61, ExecutionEnvironment.sol#L142-L317

Description: In the template `SolverBase`, `atlasSolverCall()` is called with the first parameter being `solverOp.from`. This checked to be the `owner` of the contract in `modifier safetyFirst()`.

Later on `reconcile()` is called, which currently is unauthorized: see issue "`reconcile()` can be called by anyone".

Once authorization is added to this function then it should most likely be called by the same address as `solverOp.from` to allow verification. Then it isn't practical that `solverOp.from` is the `owner`.

```
function solverMetaTryCatch(/*...*/) /*...*/ {
    // ...
    /*...*/ solverCallData = abi.encodeWithSelector(ISolverContract.atlasSolverCall.selector,
↪   solverOp.from, /*...*/ );
    (success,) = solverOp.solver.call{ gas: gasLimit, value: solverOp.value }(solverCallData);
    // ...
}
function atlasSolverCall(address sender, ...) safetyFirst(sender) /*...*/ {
    // ...
}
modifier safetyFirst(address sender) {
    // Safety checks
    require(sender == _owner, "INVALID CALLER");
    // ...
    IEscrow(_atlas).reconcile{ value: msg.value }(msg.sender, sender, shortfall);
}
```

Recommendation: Consider changing the approach to `solverOp.from` being the solver contract.

Fastlane: Solved in PR 287. Now only the solver's `solverOp.solver` address can call `reconcile()` during their `SolverOperation` phase.

Spearbit: Verified.

5.5.39 Value in revert message in `_settle()` is not obvious

Severity: Informational

Context: GasAccounting.sol#L143-L202, GasAccounting.sol#L254-L298

Description: Function `_settle()` tries to `_assign()` costs. If this fails then it `reverts` with a value that uses an updated `deposits`. This is not obvious when reading the code.

```
function _settle(/*...*/) /*...*/ {
    // ...
    if (_deposits < _claims + _withdrawals) {
        // ...
        if (_assign(winningSolver, amountOwed, true, false)) {
            revert InsufficientTotalBalance((_claims + _withdrawals) - deposits); // uses updated
↪   deposits
        }
    } else {
        // ...
        _credit(winningSolver, amountCredited);
    }
    // ...
    SafeTransferLib.safeTransferETH(bundler, _claims);
    // ...
}
```

```
function _assign(address owner, uint256 amount, bool solverWon, bool bidFind) internal returns (bool
↪   isDeficit) {
    // ...
    uint112 amt = uint112(amount);
    // ...
    if (aData.bonded < amt) {
        // ...
        if (bData.unbonding + aData.bonded < amt) {
            isDeficit = true;
            amount = uint256(bData.unbonding + aData.bonded); // contribute less to deposits ledger
            // ...
        } else {
          // ...
        }
    } else {
      // ...
    }
    bondedTotalSupply -= amount;
    deposits += amount;  // this updated value is used in revert message of _settle
}
function _credit(address owner, uint256 amount) internal {
    // ...
    bondedTotalSupply += amount;
    // ...  // no change in withdrawals
}
```

Recommendation: Suggestion: let `_assign()` return the `deficit` amount, then the `revert` message in `_settle()` is more logical:

```
  function _settle(/*...*/) /*...*/ {
-     if (_assign(winningSolver, amountOwed, true, false)) {
+     uint256 deficit = _assign(winningSolver, amountOwed, true, false));
+     if (deficit > 0) {
-         revert InsufficientTotalBalance((_claims + _withdrawals) - deposits);
+         revert InsufficientTotalBalance(deficit);
      }
  }
```

```
-  function _assign(/*...*/) /*...*/ returns (bool isDeficit) {
+  function _assign(/*...*/) /*...*/ returns (uint256 deficit) {
      // ...
      if (bData.unbonding + aData.bonded < amt) {
-         isDeficit = true;
          amount = uint256(bData.unbonding + aData.bonded);
+         deficit = amt - amount;
          // ...
      }
  }
```

Fastlane: Solved in PR 229.

Spearbit: Verified.

5.5.40 _releaseSolverLock() can be run without _trySolverLock()

Severity: Informational

Context: Escrow.sol#L97-L168

Description: In function _executeSolverOperation(), if _validateSolverOperation() fails (e.g. result !=0), then canExecute() will be false and then _trySolverLock() won't be executed.

However _releaseSolverLock() is executed. With the current code this doesn't matter because it only assigns gas costs. Once suggested changes are made this could be a problem, see the issue "Function _releaseSolverLock() doesn't undo all the actions of _trySolverLock()".

```
function _executeSolverOperation(/*...*/) /*...*/ {
    // ...
    if (result.canExecute()) {
        // ...
        (result, gasLimit) = _validateSolverOperation(dConfig, solverOp, gasWaterMark, result);
        // ...
        if (result.canExecute() && _trySolverLock(solverOp)) {
           // ...
        }
        // ...
    }
    _releaseSolverLock(solverOp, gasWaterMark, result, false, !prevalidated);
    // ...
}
```

Recommendation: Consider having a seperate function to assign the gas costs that can always be run at the end of _executeSolverOperation().

Fastlane: Solved as part of a refactor in PR 271.

Spearbit: Verified.

5.5.41 Reverting fallback() is unnecessary

Severity: Informational

Context: Escrow.sol#L478-L480

Description: Contract Escrow contains a fallback() with a revert();. This isn't necessary because the Atlas / Escrow will also revert is no fallback() is present. Perhaps its added to prevent accidentally adding another fallback() to one of the inherited contracts. In that case a comment would be useful.

```
abstract contract Escrow is AtlETH {
    fallback() external payable {
        revert();
    }
}
```

Recommendation: Consider removing the fallback() function and/or adding a comment.

Fastlane: Solved in PR 196.

Spearbit: Verified.

5.5.42 `block.timestamp` or `block.number`

Severity: Informational

Context: AtlETH.sol#L127-L169, Escrow.sol#L290-L293, AtlasVerification.sol#L130-L142, DAppApprovalTypes.sol#L14, SolverCallTypes.sol#L14, UserCallTypes.sol#L14, EscrowTypes.sol#L12

Description: The function `permit()` uses deadlines based on `block.timestamp`, however all other deadlines are based on `block.number` which might be confusing. From user's perspective, they may find it easier to know the deadline in timestamp instead of block number.

```
function permit(/*...*/) /*...*/ {
    if (deadline < block.timestamp) revert PermitDeadlineExpired();
  // ...
}
```

`AtlETH` uses `lastAccessedBlock` based on `block.number` but `_validateSolverOperation()` suggest to change to timestamp.

```
function _validateSolverOperation(
    // NOTE: Turn this into time stamp check for FCFS L2s?
    if (lastAccessedBlock == block.number) {
        result |= 1 << uint256(SolverOutcome.PerBlockLimit);
    }
}
```

For this decicion it is important to be aware that on chains like Arbitrum there can be multiple blocks within the same `block.timestamp`. Such a change will prevent `solvers` to participate in multiple blocks within the same second.

The `deadlines` in `UserOp`, `SolverOp` and `DappOp` are based on `block.number`, however no comment is made about this in the struct definitions.

```
function _validCalls(/*...*/) /*...*/ {
    // ...
      // Check if past user's deadline
      if (block.number > userOp.deadline) {
          // ...
      }
      // Check if past dapp's deadline
      if (block.number > dAppOp.deadline) {
          // ...
      }
    // ...
}
```

Recommendation: Doublecheck the correctness of the choices. Add comments to avoid confusion.

Fastlane: Solved in PR 254 by adding comments and PR 267 by removing the `permit()` function from `AtlETH`.

Spearbit: Verified.

5.5.43 `_validateSolverOperation()` uses two different ways to return a value

Severity: Informational

Context: Escrow.sol#L241-L302

Description: Function `_validateSolverOperation()` uses two different ways to return a `gasLimit` with value 0.

```
function _validateSolverOperation(/*...*/) /*...*/ returns (uint256, uint256 gasLimit){
    if (gasWaterMark < /*...*/) {
        return (result | 1 << /*...*/ , gasLimit); // gasLimit == 0
    }
    if (block.number > solverOp.deadline) {
        return ( result | 1 /*...*/, 0 ); // gasLimit == 0
    }
    // ...
}
```

Recommendation: Consider using the same pattern.

Fastlane: Solved in PR 180.

Spearbit: Verified.

5.5.44 Difference between `Sorter` and `Atlas` functions

Severity: Informational

Context: Escrow.sol#L241-L302, Sorter.sol#L59-L97, AtlasVerification.sol#L185-L221, CallVerification.sol#L10

Description: The function `Sorter - _verifySolverEligibility()` partly overlaps with checks in `_validateSolverOperation()` and `verifySolverOp()`. The overlapping parts could be combined. The differences shoud be doublechecked.

When called via `_bidFindingIteration()`: `verifySolverOp()` is done once (via `!prevalidated`) and `_validateSolverOperation()` is done twice. Operations related to `block.number` stay the same so they could move to `verifySolverOp()` to be only executed once. Also see issue "`_handleAltOpHash()` executed even in error situations".

Note: Moving the `block.number` checks would not be a good idea if `lastAccessedBlock` could be updated in the mean time. This is almost the case see issue: "Unreachable code in `_assign()`".

The names of `verifySolverOp()` and `_validateSolverOperation()` are similar which could be confusing.

Differences between functions:

_validateSolverOperation & verifySolverOp	_verifySolverEligibility
check signatures	-
check `solverOp.deadline`	-
check `solverOp.to != ATLAS`	-
check solverOp.solver == ATLAS	-
... OR solverOp.solver == address(AtlasVerification)	-
complicated formula for gas estimates	simple formula
`lastAccessedBlock == block.number`	`solverLastActiveBlock >= block.number`
`accessData[solverOp.from].bonded`	`balanceOfBonded()`
`accessData[solverOp.from].lastAccessedBlock`	`accountLastActiveBlock()`

Recommendation: Consider combining the functions of `_validateSolverOperation()`, `verifySolverOp()` and `_verifySolverEligibility()` and moving them into `library CallVerification`.

Double check all the differences between `_validateSolverOperation()`, `verifySolverOp()` and `_verifySolverEligibility()` and add any missing pieces.

Consider moving checks that only need to be executed once (e.g. `block.number` related checks) from `_validateSolverOperation()` to `verifySolverOp()`.

Double check the resulting functionality of `_validateSolverOperation()`, which will be mainly gas related and consider changing the name to a more meaningful name.

Also see issues:

- "Some functions can be moved to `AtlasVerification`".
- "`getBidValue()` is not always used".

Fastlane: Solved in PR 276.

Spearbit: Verified.

5.5.45 Some functions can be moved to `AtlasVerification`

Severity: Informational

Context: Escrow.sol#L241-L302, Escrow.sol#L394-L411

Description: `_validateSolverOperation()` is very similar to the `validate` / `verify` calls in `AtlasVerification` so they could be moved there for consistency and to free some contract size space in `Atlas`.

Function `_handleAltOpHash()` is similar to `_handleNonces()` in `AtlasVerification` so it could be moved there for consistency and to free some contract size space in `Atlas`. The function name `_handleAltOpHash()` doesn't indicate it keeps track of something.

Recommendation: Consider moving `_validateSolverOperation()` and `_handleAltOpHash()` to `AtlasVerification`. Consider changing `_handleAltOpHash()` to indicate it keeps track of something.

Also see issue "Functions `_validateSolverOperation()` and `_verifySolverEligibility()` are similar".

Fastlane: Won't fix as this requires a large refactor.

Spearbit: Acknowledged.

5.5.46 Position of `revert` in `_getBidAmount()` can be clearer

Severity: Informational

Context: Atlas.sol#L237-L295, Escrow.sol#L320-L387, ExecutionEnvironment.sol#L278

Description: Function `_getBidAmount()` does a `revert` after calling `_releaseSolverLock()`. Althought this `revert` shouldn't happen, it might just as well be done directly after the `call`. The revert doesn't return an error code, adding one might increase readability. As an unusual pattern is used, it is good to add a comment.

```
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    key.bidFind = true;
    // ...
    /*...*/ _getBidAmount(/*...*/,key) /*...*/
    // ...
}
function _getBidAmount(/*...*/) /*...*/ {
    // ...
    data = abi.encodeWithSelector( /*...*/ solverMetaTryCatch.selector, /*...*/);
    // ...
    (success, data) = key.executionEnvironment.call{ value: solverOp.value }(data);
    _releaseSolverLock(solverOp, gasWaterMark, result, true, true);
    if (success) {
        revert();
    }
    // ...
}
function solverMetaTryCatch(
 if (_bidFind()) {
revert AtlasErrors.BidFindSuccessful(netBid);
```

Recommendation: Consider moving the `revert` and adding a comment, and possibly a custom error. For example in the following way:

```
  function _getBidAmount(/*...*/) /*...*/ {
      // ...
      (success, data) = key.executionEnvironment.call{ value: solverOp.value }(data);
-     _releaseSolverLock(solverOp, gasWaterMark, result, true, true);
      if (success) {  // should never happen because call will revert with BidFindSuccessful() if
↪   successful
          revert ShouldNeverHappen();
      }
+     _releaseSolverLock(solverOp, gasWaterMark, result, true, true);
      // ...
  }
```

Fastlane: Solved in PR 225. `_releaseSolverLock()` is no longer done inside `_getBidAmount()`, so the unreachable revert now happens directly after the call. Comment and clearer name added for clarity.

Spearbit: Verified.

5.5.47 Function `_validateSolverOperation()` doesn't need parameter `result`

Severity: Informational

Context: Escrow.sol#L119-L129, Escrow.sol#L241-L302, Escrow.sol#L339-L342

Description: In both `_executeSolverOperation()` and `_getBidAmount()` it is verified that `result==0` before calling `_validateSolverOperation()`. So `result` doesn't have to be supplied to `_validateSolverOperation()`.

Note: `return (result |= 1 ...);` doesn't require the `=` because the updated `result` isn't used after the `return`.

```
function _executeSolverOperation(
    if (result.canExecute()) {
         // ...
        (result, gasLimit) = _validateSolverOperation(/*...*/, result);
    }
    // ...
}
function _getBidAmount(
    // ...
    if (!result.canExecute()) return 0;
    (result, gasLimit) = _validateSolverOperation(..., result);
    // ...
}
function _validateSolverOperation(...,  uint256 result) /*...*/ {
    if (/*...*/) {
        return (result | 1 << uint256(SolverOutcome.UserOutOfGas), gasLimit);
    }
    if (/*...*/) {
        return (result | 1 << uint256(dConfig.callConfig.allowsTrustedOpHash()
                  ? uint256(SolverOutcome.DeadlinePassedAlt) : uint256(SolverOutcome.DeadlinePassed) ),
↪   0  );
    }
    if (/*...*/) {
        return (result |= 1 << uint256(SolverOutcome.CallValueTooHigh), gasLimit);
    }
    if (lastAccessedBlock == block.number) {
        result |= 1 << uint256(SolverOutcome.PerBlockLimit);
    }
    if (gasCost > solverBalance) {
        result |= 1 << uint256(SolverOutcome.InsufficientEscrow);
    }
    return (result, gasLimit);
}
```

Recommendation: Consider removing the argument `result` from `_validateSolverOperation()`. After this the function `_validateSolverOperation()` itself can be simplied because most of the time a `return` is done when an error occurs. In that case `result |` / `result |=` can be removed.

Fastlane: Solver in PR 183.

Spearbit: Verified.

5.5.48 Typos

Severity: Informational

Context: See each case below.

Description: There are typos in the following locations:

- Escrow.sol#L86: SovlerOperation → SolverOperation.

```
/// @notice Attempts to execute a SovlerOperation and determine if it wins the auction.
```

- GasAccounting.sol#L219-L220: SovlerOperation → SolverOperation

```
/// @dev Calculates the gas used for the SovlerOperation and adjusts the solver's escrow balance
↪  accordingly.
/// @param solverOp The current SovlerOperation for which to account
```

- DAppIntegration.sol#L36: arent → aren't

```
// processed in any order so long as they arent duplicated and
```

- EscrowBits.sol#L68: is → if

```
// returns true is solver doesn't get to bypass the refund.
```

Recommendation: Correct each typo listed above.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.49 Store "*magic numbers*" as constants

Severity: Informational

Context: See each case below.

Description: There are magic numbers used throughout the codebase, which can make the code harder to understand compared to using constant variables. Some usages of magic numbers include:

- AtlasVerification.sol#L550:

```
IAccount(userOp.from).validateUserOp{ gas: 30_000 }(userOp, _getProofHash(userOp), 0) == 0;
```

- GasAccounting.sol#L238:

```
uint256 gasUsed = (gasWaterMark - gasleft() + 5000) * tx.gasprice;
```

- GasAccounting.sol#L244:

```
gasUsed = (gasUsed + ((gasUsed * SURCHARGE) / 10_000_000));
```

- GasAccounting.sol#L273:

```
gasRemainder += ((gasRemainder * SURCHARGE) / 10_000_000);
```

- GasAccounting.sol#L289:

```
uint256 netGasSurcharge = (_claims * SURCHARGE) / 10_000_000;
```

- Escrow.sol#L269-L270:

```
gasLimit = (100) * (solverOp.gas < dConfig.solverGasLimit ? solverOp.gas :
↪  dConfig.solverGasLimit)
           / (100 + _SOLVER_GAS_BUFFER) + _FASTLANE_GAS_BUFFER;
```

- SafetyLocks.sol#L42:

```
claims = rawClaims + ((rawClaims * SURCHARGE) / 10_000_000);
```

- Storage.sol#L58-L59:

```
uint256 internal _solverCalledBack = 1 << 161;
uint256 internal _solverFulfilled = 1 << 162;
```

- AtlasVerification.sol#L409-L410:

```
uint256 bitmapIndex = ((nonce - 1) / 240) + 1; // +1 because highestFullBitmap initializes at 0
uint256 bitmapNonce = ((nonce - 1) % 240); // 1 -> 0, 240 -> 239. Needed for shifts in bitmap.
```

- Atlas.sol#L51:

```
uint256 gasMarker = gasleft(); // + 21_000 + (msg.data.length * _CALLDATA_LENGTH_PREMIUM);
```

- Escrow.sol#L448:

```
if (success) { return uint256(0); }
```

- EscrowBits.sol#L48-L61:

```
return (result == 0);
```

- Simulator.sol#L91:

```
return (false, Result.Unknown, uint256(type(SolverOutcome).max) + 1);
```

- Storage.sol#L84-L86:

```
claims = type(uint256).max;
withdrawals = type(uint256).max;
deposits = type(uint256).max;
```

- SafetyLocks.sol#L93-L95:

```
claims = type(uint256).max;
withdrawals = type(uint256).max;
deposits = type(uint256).max;
```

- GasAccounting.sol#L34:

```
if (_deposits != type(uint256).max) {
```

- GasAccounting.sol#L166:

```
if (amount > type(uint112).max) revert ValueTooLarge();
```

- GasAccounting.sol#L214:

```
if (amount > type(uint112).max) revert ValueTooLarge();
```

- AtlETH.sol#L111:

```
if (allowed != type(uint256).max) allowance[from][msg.sender] = allowed - amount;
```

- AtlasVerification.sol#L387:

```
if (nonce > type(uint128).max - 1) {
```

- CallBits.sol#L103-L181:

```
sequenced = (callConfig & 1 << uint32(CallConfigIndex.UserNoncesSequenced) != 0);
// ...
return (callConfig & 1 << uint32(CallConfigIndex.ExPostBids) != 0);
```

Recommendation: Consider defining new constant variables, and use those in place of each "*magic number*".

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.50 Do safety checks as early as possible

Severity: Informational

Context: ExecutionEnvironment.sol#L173-L176, AtlasVerification.sol#L70

Description: Safety checks should be done as early as possible. It makes it easy to remove certain assumptions when going through the code, and improves readability.

Recommendation: As an example, `validCalls()` can be changed to the following. We can afford to calculate `userOpHash` later because the caller (Atlas.sol) only cares of `userOpHash` when the call is valid.

```
function _validCalls(
    DAppConfig calldata dConfig,
    UserOperation calldata userOp,
    SolverOperation[] calldata solverOps,
    DAppOperation calldata dAppOp,
    uint256 msgValue,
    address msgSender,
    bool isSimulation
)
    internal
    returns (bytes32 userOpHash, ValidCallsResult)
{

    {
        // Check user signature
        if (!_verifyUser(dConfig, userOp, msgSender, isSimulation)) {
            return (userOpHash, ValidCallsResult.UserSignatureInvalid);
        }

        // Check if past user's deadline
        if (block.number > userOp.deadline) {
            if (userOp.deadline != 0 && !isSimulation) {
                return (userOpHash, ValidCallsResult.UserDeadlineReached);
            }
        }

        // Check if past dapp's deadline
        if (block.number > dAppOp.deadline) {
            if (dAppOp.deadline != 0 && !isSimulation) {
                return (userOpHash, ValidCallsResult.DAppDeadlineReached);
            }
        }

        // Check gas price is within user's limit
        if (tx.gasprice > userOp.maxFeePerGas) {
            return (userOpHash, ValidCallsResult.GasPriceHigherThanMax);
        }

        // Check that the value of the tx is greater than or equal to the value specified
        if (msgValue < userOp.value) {
            return (userOpHash, ValidCallsResult.TxValueLowerThanCallValue);
        }
    }

    {
        uint256 solverOpCount = solverOps.length;

        // Check solvers not over the max (253)
        if (solverOpCount > MAX_SOLVERS) {
            return (userOpHash, ValidCallsResult.TooManySolverOps);
        }
        // Verify a solver was successfully verified.
        if (solverOpCount == 0) {
            if (!dConfig.callConfig.allowsZeroSolvers()) {
                return (userOpHash, ValidCallsResult.NoSolverOp);
            }

            if (dConfig.callConfig.needsFulfillment()) {
                return (userOpHash, ValidCallsResult.NoSolverOp);
            }
        }
    }

    // Verify that the calldata injection came from the dApp frontend
    // and that the signatures are valid.

    // CASE: Solvers trust app to update content of UserOp after submission of solverOp
    if (dConfig.callConfig.allowsTrustedOpHash()) {
        userOpHash = userOp.getAltOperationHash();

        // SessionKey must match explicitly - cannot be skipped
        if (userOp.sessionKey != dAppOp.from && !isSimulation) {
            return (userOpHash, ValidCallsResult.InvalidAuctioneer);
        }

        // msgSender must be userOp.from or userOp.sessionKey / dappOp.from
        if (msgSender != dAppOp.from && msgSender != userOp.from && !isSimulation) {
            return (userOpHash, ValidCallsResult.InvalidBundler);
        }
    } else {
        userOpHash = userOp.getUserOperationHash();
    }

    {
        // bypassSignatoryApproval still verifies signature match, but does not check
        // if dApp approved the signor.
        (bool validAuctioneer, bool bypassSignatoryApproval) = _verifyAuctioneer(dConfig, userOp,
↪   solverOps, dAppOp);

        if (!validAuctioneer && !isSimulation) {
            return (userOpHash, ValidCallsResult.InvalidAuctioneer);
        }

        // Check dapp signature
        (bool validDAppOp, ValidCallsResult result) =
            _verifyDApp(dConfig, dAppOp, msgSender, bypassSignatoryApproval, isSimulation);

        if (!validDAppOp) {
            return (userOpHash, result);
        }
    }

    // Some checks are only needed when call is not a simulation
    if (isSimulation) {
        // Add all solver ops if simulation
        return (userOpHash, ValidCallsResult.Valid);
    }

    if (userOpHash != dAppOp.userOpHash) {
        return (userOpHash, ValidCallsResult.OpHashMismatch);
    }
    return (userOpHash, ValidCallsResult.Valid);
}
```

Fastlane: Solved in PR 248.

Spearbit: Verified.

5.5.51 Consider allowing arbitrary calls from the `ExecutionEnvironment`

Severity: Informational

Context: ExecutionEnvironment.sol

Description: Although the `ExecutionEnvironment` is somewhat similar to a smart contract wallet, its non-Atlas functionality is limited to the `withdrawERC20()` and `withdrawEther()` functions. Technically the Atlas functions (e.g. `userWrapper()`) can execute arbitrary logic, however, this would depend on how the relevant `DAppControl` guards these calls.

In niche situations, more functionality may be desired by users. For example, if an `ExecutionEnvironment` becomes eligible for an airdrop, it may not be possible to claim the airdrop with any of the existing functions.

Recommendation: Consider adding a new function in the `ExecutionEnvironment` that allows the user to make an arbitrary call. This function should only be callable by the `_user()`, and should also check against the `isUnlocked()` guard in the Atlas contract.

If the suggestion of issue "`Mimic` can be optimized" is followed, then some additonal changes are required.

Fastlane: WontFix for now. This is a good suggestion, but is something we will have to consider for Atlas v2 as it's potentially quite a big change.

Spearbit: Acknowledged

5.5.52 `bypassSignatoryApproval` isn't clear

Severity: Informational

Context: AtlasVerification.sol#L223-L253

Description: The use of `bypassSignatoryApproval` is not easy to understand: `bypassSignatoryApproval` is used when the `DappControl` doesn't want/have to verify the signer of the `dAppOp` e.g. if the `DappControl` isn't the responsible party.

```
/// @return bypassSignatoryApproval A boolean indicating if the signatory approval check should be
↪   bypassed.
function _verifyAuctioneer(/*...*/) /*...*/ returns (/*...*/, bool bypassSignatoryApproval) {
    // ...
}
```

Recommendation: Consider changing the name to something like `isDappControlResponsible`.

Fastlane: Solved in PR 227.

Spearbit: Verified.

5.5.53 Bool return value looses error information

Severity: Informational

Context: AtlasVerification.sol#L231-L253, AtlasVerification.sol#L307-L377, AtlasVerification.sol#L532-L575

Description: The function below all return different types of data, but all want to expose error codes. _verifyDApp() and _verifyUser() only return bool which looses information about the error. Functions _verifyAuctioneer() that return both a bool an and error. The error would be sufficient.

```
function _verifyDApp      (/*...*/) /*...*/ returns (bool /*valid*/, ValidCallsResult) { }
function _verifyUser      (/*...*/) /*...*/ returns (bool /*valid*/) { }
function _verifyAuctioneer(/*...*/) /*...*/ returns (bool valid, bool bypassSignatoryApproval) { }
```

Recommendation: Consider to let all functions return ValidCallsResult and no bool.

Fastlane: Solved in PR 212.

Spearbit: Verified.

5.5.54 Inaccurate comment of _nonceUsedInBitmap()

Severity: Informational

Context: AtlasVerification.sol#L667-L674, AtlasVerification.sol#L386-L447

Description: The relevant values for the nonce parameter of _nonceUsedInBitmap() are 0 - 239 as can be seen in function _handleNonces(). So the comment of _nonceUsedInBitmap() isn't accurate.

```
/// @dev Only accurate for nonces 1 - 240 within a 256-bit bitmap.
// ...
function _nonceUsedInBitmap(uint256 bitmap, uint256 nonce) internal pure returns (bool) {
    return (bitmap & (1 << nonce)) != 0;
}
function _handleNonces(address account, uint256 nonce, bool async, bool isSimulation) internal returns
↪  (bool) {
    // ...
    uint256 bitmapNonce = ((nonce - 1) % 240); //  0 <= bitmapNonce <= 239
    // ...
    if (_nonceUsedInBitmap(bitmap, bitmapNonce)) {
        // ...
    }
    // ...
}
```

Recommendation: Change the comment to:

```
- /// @dev Only accurate for nonces 1 - 240 within a 256-bit bitmap.
+ /// @dev Only accurate for nonces 0 - 239 within a 256-bit bitmap.
```

Fastlane: Solved in PR 259 due to a refactor of the nonces.

Spearbit: Verified.

5.5.55 `manuallyUpdateNonceTracker()` can miss blocks that are not completely filled

Severity: Informational

Context: AtlasVerification.sol#L646-L665

Description: The function `manuallyUpdateNonceTracker()` steps forward 10 positions and then searches backwards for the first `FULL_BITMAP` and stops when it finds one. However there might be intermediate block that are not completely filled. This deviates from the rest of the logic and would break invariants.

```
function manuallyUpdateNonceTracker(address account) external {
    // ...
    // Checks the next 10 bitmaps for a higher full bitmap
    uint128 nonceIndexToCheck = nonceTracker.highestFullAsyncBitmap + 10;
    for (; nonceIndexToCheck > nonceTracker.highestFullAsyncBitmap; nonceIndexToCheck--) {
        bytes32 bitmapKey = keccak256(abi.encode(account, nonceIndexToCheck));
        nonceBitmap = nonceBitmaps[bitmapKey];
        if (nonceBitmap.bitmap == FULL_BITMAP) {
            nonceTracker.highestFullAsyncBitmap = nonceIndexToCheck;
            break;
        }
    }
    // ...
}
```

Recommendation: See the suggestion of the issue "`Nonce` logic is complicated".

Fastlane: Solved in PR 259. This function has been removed as part of the nonce logic simplification.

Spearbit: Verified.

5.5.56 Function name `manuallyUpdateNonceTracker()` not clear

Severity: Informational

Context: AtlasVerification.sol#L646-L665

Description: Function `manuallyUpdateNonceTracker()` only works for `async (bitmap) nonces` and not for `sequential nonces`. This isn't clear from the function name.

```
function manuallyUpdateNonceTracker(address account) external {
    // ...
    uint128 nonceIndexToCheck = nonceTracker.highestFullAsyncBitmap + 10;
    // ...
}
```

Recommendation: Consider changing the function name to somethink like:

```
- function manuallyUpdateNonceTracker(address account) external {
+ function manuallyUpdateAsyncNonceTracker(address account) external {
```

Fastlane: Solved in PR 259. This function has been removed as part of the nonce logic simplification.

Spearbit: Verified.

5.5.57 Incorrect comment in Mimic

Severity: Informational

Context: Mimic.sol#L7

Description: This comment hints that user can just use an EOA with Atlas:

```
0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB is standin for the user's EOA address
```

User can also have a smart contract account which works well with Atlas.

Recommendation: Update the comment as:

```
0xbBbBBBBbbBBBbbbBbbBbbbbBBbBbbbbBbBbbBBbB is standin for the userOp.from address
```

The recommendation in issue "Mimic can be optimized" fixes this issue as well.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.58 VERIFICATION can be typed to AtlasVerification

Severity: Informational

Context: Storage.sol#L19

Description: VERIFICATION is always set to AtlasVerification contract. To remove any ambiguity, change its type to AtlasVerification instead of address.

Recommendation: Change VERIFICATION type from address to AtlasVerification.

Fastlane: Solved in PR 233.

Spearbit: Verified.

5.5.59 Rearranging terms will achieve higher precision

Severity: Informational

Context: SafetyLocks.sol#L42, GasAccounting.sol#L273

Description: The following computations can be rearranged to achieve higher precision for claims and gasRemainder:

```
claims = rawClaims + ((rawClaims * SURCHARGE) / 10_000_000);
```

```
gasRemainder += ((gasRemainder * SURCHARGE) / 10_000_000);
```

Recommendation: Update them as:

- `claims = rawClaim * (10_000_000 + SURCHARGE) / 10_000_000`
- `gasRemainder = gasRemainder * (10_000_000 + SURCHARGE) / 10_000_000`

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.60 Some `...get...Hash` functions don't have `..._TYPE_HASH`

Severity: Informational

Context: AtlasVerification.sol#L481-L497, AtlasVerification.sol#L580-L597, AtlasVerification.sol#L273-L291, CallVerification.sol#L11-L17

Description: The functions `_getProofHash()` (2x) and `_getSolverHash()` include a `..._TYPE_HASH` in the hashed data. While the similar functions `getUserOperationHash()` and `getAltOperationHash()`.

Including `..._TYPE_HASH` helps to make the data unique and prevent overlaps with other hashed data. Both `_getProofHash()` for `userOp` and `getUserOperationHash()` do a signature over 12 fields. Luckily the types per field are different, otherwise there might a a collision between the two hashes.

```
 function _getProofHash(UserOperation memory userOp) internal pure returns (bytes32 proofHash) {
    proofHash = keccak256(abi.encode(USER_TYPE_HASH, /*...*/));
}
function _getProofHash(DAppOperation memory approval) internal pure returns (bytes32 proofHash) {
     proofHash = keccak256(abi.encode(DAPP_TYPE_HASH, /*...*/));
}
function _getSolverHash(SolverOperation calldata solverOp) internal pure returns (bytes32 solverHash) {
    return keccak256(abi.encode(SOLVER_TYPE_HASH, /*...*/ ));
}
function getUserOperationHash(UserOperation memory userOp) internal pure returns (bytes32 userOpHash) {
    userOpHash = keccak256(abi.encode(userOp)); // no ..._TYPE_HASH
}
function getAltOperationHash(UserOperation memory userOp) internal pure returns (bytes32 altOpHash) {
    altOpHash = keccak256(abi.encodePacked(userOp.from, /*...*/)); // no ..._TYPE_HASH
}
```

Recommendation: Consider adding a `..._TYPE_HASH` for `getUserOperationHash()` and `getAltOperationHash()`.

Also see the issue "Functions `_getProofHash()` and `getUserOperationHash()` are very similar".

Fastlane: Solved in PR 251.

Spearbit: Verified.

5.5.61 Functions `_getProofHash()` and `getUserOperationHash()` are very similar

Severity: Informational

Context: AtlasVerification.sol#L580-L597, CallVerification.sol#L11-L17

Description: There are two ways to retrieve the hash of the `userOp`, via `_getProofHash()` and `getUserOperationHash()`. Function `getUserOperationHash()` also includes the `userOp.signature`, which isn't necessary because the hash of the rest of the data is already unique. The signature of a signature increases complexity. Having two very similar functions also increases code size and complexity.

Note: also see the issue "Some `...get...Hash` functions don't have `..._TYPE_HASH`" for another difference.

```
function _getProofHash(UserOperation memory userOp) internal pure returns (bytes32 proofHash) {
    proofHash = keccak256(
        abi.encode(
            USER_TYPE_HASH,
            userOp.from,
            userOp.to,
            userOp.value,
            userOp.gas,
            userOp.maxFeePerGas,
            userOp.nonce,
            userOp.deadline,
            userOp.dapp,
            userOp.control,
            userOp.sessionKey,
            keccak256(userOp.data)
        ) // userOp.signature is not included
    );
}
function getUserOperationHash(UserOperation memory userOp) internal pure returns (bytes32 userOpHash) {
    userOpHash = keccak256(abi.encode(userOp));
}
```

Recommendation: Consider only using `_getProofHash()`.

Fastlane: Solved in PR 251. There are two types of hashes for a userOp, and "userOp hash" and a "userOp-Payload hash". The payload hash is for signing, and the userOp hash is for references to the userOp from the solverOp and dAppOp. The payload is always the full message, and the userOp hash is either the full or the trusted op hash, based on what the allowsTrustedOpHash setting is set to.

Spearbit: Verified.

5.5.62 `_get...Hash` functions use different name patterns

Severity: Informational

Context: AtlasVerification.sol#L481-L497, AtlasVerification.sol#L580-L597, AtlasVerification.sol#L273-L291

Description: The function `_getProofHash()` if overloaded for `UserOperation`, but for `SolverOperation` there is a different function name. It would be clearer to use the same pattern.

```
function _getProofHash(DAppOperation /*...*/) /*...*/ {}
function _getProofHash(UserOperation /*...*/) /*...*/ {}
function _getSolverHash(SolverOperation /*...*/) /*...*/ {}
```

Recommendation: Consider including the struct name in the function name:

```diff
- function _getProofHash(DAppOperation /*...*/) /*...*/ {}
+ function _getDAppOpHash(DAppOperation /*...*/) /*...*/ {}
- function _getProofHash(DAppOperation /*...*/) /*...*/ {}
+ function _getUserOpHash(UserOperation /*...*/) /*...*/ {}
- function _getSolverHash(SolverOperation /*...*/) /*...*/ {}
+ function _getSolverOpHash(SolverOperation /*...*/) /*...*/ {}
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.63 `bitmap256` uses a different pattern

Severity: Informational

Context: AtlasVerification.sol#L620-L641

Description: Most functions in `AtlasVerification` use `uint256 bitmap`. However function `getNextNonce()` uses `uint256 bitmap256;`. It would be more consistent to use the same pattern everywhere.

```
function getNextNonce(address account, bool sequenced) external view returns (uint256) {
    // ...
    uint256 bitmap256;
    // ...
}
```

Recommendation: Consider changing to `uint256 bitmap`:

```
- uint256 bitmap256;
+ uint256 bitmap;
```

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.64 Error `DAppSignatureInvalid` in `_verifyDApp()` is not specific

Severity: Informational

Context: AtlasVerification.sol#L307-L377

Description: The function `_verifyDApp()` returns with error `DAppSignatureInvalid` if a `SignatoryCheck` fails. This error is not specific and difficult to trace back to the cause.

```
function _verifyDApp(/*...*/) /*...*/ {
    // ...
    if (!bypassSignatoryCheck) {
        return (false, ValidCallsResult.DAppSignatureInvalid); // not specific
    }
    // ...
}
```

Recommendation: Consider using a more specific error in `_verifyDApp()` for example `DappNotEnabled`.

Fastlane: Solved in PR 231.

Spearbit: Verified.

5.5.65 Function `_verifyDApp()` accesses `signatories[]` directly

Severity: Informational

Context: AtlasVerification.sol#L307-L377, DAppIntegration.sol#L186-L189

Description: The function `_verifyDApp()` accesses the array `signatories[]` directly. This array is part of the contract `DAppIntegration`. This exposes the implementation details.

```
function _verifyDApp(
    // ...
    if (!signatories[keccak256(abi.encodePacked(dAppOp.control, msgSender))]) { /*...*/ }
    // ...
    if (!signatories[keccak256(abi.encodePacked(dAppOp.control, dAppOp.from))]) { /*...*/ }
    // ...
}
function isDAppSignatory(address dAppControl, address signatory) external view returns (bool) {
    bytes32 signatoryKey = keccak256(abi.encodePacked(dAppControl, signatory));
    return signatories[signatoryKey];
}
```

Recommendation: Consider using function `isDAppSignatory()` to access the array `signatories[]`. This can be done in the following way:

```
- if (!signatories[keccak256(abi.encodePacked(dAppOp.control, /*...*/))]) { /*...*/ }
+ if (!isDAppSignatory(dAppOp.control, /*...*/)) { /*...*/ }
```

Fastlane: Solved in PR 195.

Spearbit: Verified.

5.5.66 Contract `AtlasVerification` doesn't import `ECDSA`

Severity: Informational

Context: AtlasVerification.sol#L4-L22

Description: Contract `AtlasVerification` uses `ECDSA` from `EIP712.sol`. However the latest version of `EIP712.sol` doesn't import `ECDSA` anymore, so when an upgrade is done `AtlasVerification` doesn't compile anymore. Additionally it is also clearer to directly import `ECDSA`. There are the different versions:

- EIP712 v5.0
- EIP712 v4.9

```
import "openzeppelin-contracts/contracts/utils/cryptography/EIP712.sol";
contract AtlasVerification is EIP712, /*...*/ {
    using ECDSA for bytes32;
    // ...
}
```

Recommendation: Consider adding an import of `ECDSA` to `AtlasVerification`.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.67 Comment of `execute()` is incorrect

Severity: Informational

Context: Atlas.sol#L42-L96, Atlas.sol#L98-L151, Atlas.sol#L237-L295

Description: The comment of the last return value of `execute()` differs from the code.

```
function metacall(/*...*/) /*...*/ {
    // ...
    try this.execute{ /*...*/ }(/*...*/)
    returns (/*...*/ , uint256 winningSolverIndex) {
        // ...
    } // ...
}

/// @return uint256 The solver outcome bitmap ==> should be winningSolverIndex
function execute(/*...*/) /*...*/ {
    // ...
    (/*...*/, key) = _bidFindingIteration(/*...*/);
    // ...
    return (/*...*/, uint256(key.solverOutcome));
}
function _bidFindingIteration(
    // ...
    key.solverOutcome = uint24(bidPlaceholder); // is winningSolverIndex
    return (/*...*/, key);
    // ...
}
```

Recommendation: Change the comment of `execute()` to something like this:

```
- /// @return uint256 The solver outcome bitmap
+ /// @return uint256 The winning Solver Index
```

Fastlane: Solved by refactoring.

Spearbit: Verified.

5.5.68 Reuse of variables is confusing

Severity: Informational

Context: Atlas.sol#L237-L295, Escrow.sol#L97-L168, ExecutionEnvironment.sol#L142-L317

Description: Sometimes variables are used for multiple purpuses, which is confusing when reading the code and also allows for errors in future code updates. This is most likely done to prevent "*stack too deep*" issues. The following examples have been found:

- `key.solverOutcome` which is used as an error code and as the index for the winning solver.
- `endBalance` which is used as the end token balance and as the remaining ETH balance.

```
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    (auctionWon, key) = _executeSolverOperation( /*...*/ );
    // ...
    key.solverOutcome = uint24(bidPlaceholder);
    // ...
}
function _executeSolverOperation(
    // ...
    key.solverOutcome = uint24(result); // error code
    // ...
    return (false, key);
}
```

```
function solverMetaTryCatch(
    // ...
    endBalance = etherIsBidToken ? endBalance : address(this).balance;
    if (endBalance > 0) {
         IEscrow(atlas).contribute{ value: endBalance }();
    }
    // ...
}
```

Recommendation: Check if the dual use is really necessary. Consider adding more comments about the dual use. Consider expressing the dual use in the variable name.

Fastlane: Solved by refactoring.

Spearbit: Verified.

5.5.69 `bidFind` state is handled in a different way

Severity: Informational

Context: Atlas.sol#L237-L295

Description: Within function `_bidFindingIteration()` a seperate state is maintained, the `bidFind` state. This could be integrated with other state mechanisms to simplify the code.

```
function _bidFindingIteration(/*...*/) /*...*/ {
    // ...
    key.bidFind = true;
    // ...
    bidPlaceholder = _getBidAmount(dConfig, userOp, solverOps[i], returnData, key);
    // ...
    key.bidFind = false;
    // ...
}
```

Recommendation: Consider making the `BidFind` phases explicit, for example in the following way:

```
  enum ExecutionPhase {
      Uninitialized,
      PreOps,
      UserOperation,
+     BidFindPreSolver,
+     BidFindSolverOperations,
+     BidFindPostSolver,
      PreSolver,
      SolverOperations,
      PostSolver,
      HandlingPayments,
      PostOps,
      Releasing
  }
```

Also see:

- Locking mechanism is complicated

Fastlane: No real need for extra phases after refactoring.

Spearbit: Acknowledged.

5.5.70 Inconsistent way to call `CallBits`

Severity: Informational

Context: Atlas.sol#L182-L186

Description: `CallBits` functions are usually called as `CallBits.fn(self)`, but in the highlighted `if` condition, it's called as `self.fn()`.

Recommendation: Consider following the same pattern everywhere.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.71 Remove unused variables, constants and imports

Severity: Informational

Context: ExecutionBase.sol#L17, ExecutionBase.sol#L12, ChainlinkDAppControl.sol#L174, ChainlinkDAppControl.sol#L13, SolverBase.sol#L66-L67, SafetyLocks.sol#L5, Simulator.sol#L15-L17, V2DAppControl.sol#L35, V2DAppControl.sol#L41-L42

Description: Highlighted variables, constants and imports (including test imports) aren't used.

Recommendation: Remove these variables, constants and imports.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.72 Parenthesis can be used to remove ambiguity on the order of operations

Severity: Informational

Context: CallBits.sol#L103-L180

Description: The code highlighted leaves some room for ambiguity on the order of operations for the code reader.

Recommendation: Parenthesis can be used to remove this ambiguity:

```diff
- return (callConfig & 1 << uint32(CallConfigIndex.ExPostBids) != 0);
+ return (callConfig & (1 << uint32(CallConfigIndex.ExPostBids)) != 0);
```

Fastlane: Solved in PR 192.

Spearbit: Verified.

5.5.73 Prefer `control` naming over `controller`

Severity: Informational

Context: SafetyBits.sol#L69, CallBits.sol#L11

Description: `control` and `controller` is used to refer to the same actor in the system at different places.

Recommendation: Prefer using `control` wherever `controller` is used.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.74 Use `abi.encodeCall` instead of `abi.encodeWithSelector`

Severity: Informational

Context: Escrow.sol#L54

Description: `abi.encodeCall` does compile-time type check on function arguments. Hence, it should be preferred over `abi.encodeWithSelector`.

Recommendation: Replace all uses of `abi.encodeWithSelector` with `abi.encodeCall`. The highlighted instance is just one example.

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.75 Outdated comment for `_bidKnownIteration()`

Severity: Informational

Context: Atlas.sol#L320

Description: `_bidKnownIteration()` code doesn't match with this comment.

```
// valid solverOps are packed from left of array - break at first invalid solverOp
```

The loop doesn't break, and it iterates through the entire `solverOps` array.

Recommendation: Remove the comment.

Fastlane: Solved due to refactoring. That comment may have been left behind from an older version. Previously, we would verify all `solverOps` in `validateCalls`, and exclude any from the array that didn't pass verification. It refers to that previous assumption that we could iterate through the array and if we hit an invalid `solverOp`, there would be no point iterating further.

But thats changed now, `solverOps` are individually verified and executed in `_bidKnownIteration()` mode.

Spearbit: Verified.

5.5.76 Mix of `require` and `revert` statements

Severity: Informational

Context: contracts

Description: There's a mix of `require`, `revert` with strings and `revert` with custom error statements in the code.

Recommendation: Prefer one style over the other. When changing between `revert` and `require` statements, ensure that the boolean conditions are flipped correctly. Also, it's likely that the next Solidity version introduces `require` statements with custom errors (see Solidity PR 14913).

Fastlane: Solved in PR 155.

Spearbit: Verified.

5.5.77 `Escrow` can inherit `IEscrow`

Severity: Informational

Context: Escrow.sol, IEscrow.sol

Description: `IEscrow` is an interface declaring functions defined in `Escrow` contract. To make this concrete and to ensure that `Escrow` contract confirms to the interface, inheritance can be used.

Recommendation: Inherit `IEscrow` in `Escrow`:

```
abstract contract Escrow is AtlETH, IEscrow {
```

Fastlane: The different interface files are combined in a single `IAtlas` interface in PR 278. Making `Atlas` inherit from `IAtlas` is not done as the refactor is quite large.

Spearbit: Acknowledged.

5.5.78 Locking mechanism is complicated

Severity: Informational

Context: LockTypes.sol#L27-L37, SafetyLocks.sol#L34-L96, GasAccounting.sol#L208-L246

Description: There are three different locking mechanisms used by `Atlas`:

- `_setAtlasLock()` / `_releaseAtlasLock()`, which keeps track of the `ExecutionEnvironment`.
- `_trySolverLock()` / `_releaseSolverLock()`, which keeps track of the current `solver`.
- `_buildEscrowLock()` / `key.lockState` which keeps track of the current phase.

The phase information is tracked in the `ExecutionEnvironment`, which is not reliable and is not accessible from `Atlas`. The `ExecutionEnvironment` also uses sub phases (e.g. `BidFind` is `true` or `false`). The functions for each phase are called from different locations. The role and phase limitation are not always enforced. Having different mechanisms is difficult to understand and maintain.

See the following related issues:

- `validControl` / `onlyAtlasEnvironment` are not effective in `delegatecall` situation
- Code duplications for call to `_allocateValue()`
- Passsing of `key` can be simplified
- `bidFind` state is handled in a different way
- Winning solver doesn't get gas costs `_assign()`ed
- Function `_releaseSolverLock()` doesn't undo all the actions of `_trySolverLock()`
- `_releaseSolverLock()` can be run without `_trySolverLock()`
- `callIndex` incremented twice
- `userWrapper()` does not always need `forward()` data
- Checks for `solverCalledBack` don't cover all situations
- `reconcile()` creates `deposits` out of thin air
- Moving `validateBalances()` to `Atlas`
- `Borrow()`s after `validateBalances()`
- `atlasSolverCall()` doesn't check caller
- `factoryWithdrawERC20()` and `factoryWithdrawEther()` not used

Recommendation: Consider keeping track of the roles and the current phase as well as calling the phase functions from a central place in `Atlas`. With transient storage these costs of the state tracking will be more manageable. Consider making the `BidFind` phases explicit:

```
  enum ExecutionPhase {
      Uninitialized,
      PreOps,
      UserOperation,
+     BidFindPreSolver,
+     BidFindSolverOperations,
+     BidFindPostSolver,
      PreSolver,
      SolverOperations,
      PostSolver,
      HandlingPayments,
      PostOps,
      Releasing
  }
```

As there are several different locks, it might be useful to change the names for both the function names and the `lock` variables, for example:

- `_setAtlasLock()` → `_setEELock()` or `setExecutionEnvironmentLock()`.
- `lock` → `EELock` or `executionEnvironmentLock`.

Fastlane: Solved in PR 227 due to refactoring.

Spearbit: Verified.

5.5.79 Return values of `execute()` can be simplified

Severity: Informational

Context: Atlas.sol#L70-L86

Description: The only relevant information returned from `execute()` is the fact that the `auction` was won and who the `winning` solver was. This can be expressed in one variable, which would simplify the code.

```
function metacall(/*...*/) returns (bool auctionWon) {
    // ...
    try this.execute{ /*...*/ }(/*...*/)  returns (bool _auctionWon, uint256 winningSolverIndex) {
        auctionWon = _auctionWon;
        (/*...*/) = _settle({winningSolver: auctionWon ? solverOps[winningSolverIndex].from :
↪   msg.sender, /*...*/ });
        emit MetacallResult( /*...*/, auctionWon ? solverOps[winningSolverIndex].from : address(0),
↪   /*...*/);
    }
    // ...
}
```

Note: if `auctionWon == false` then `winningSolverIndex` still contains an error code set in `_executeSolverOperation()`, due to this issue: "Reuse of variables is confusing".

Recommendation: Consider returning the `winningSolver` from `execute()`. And use a value of `0` if there is no winning solver. The code could then be simplified to:

```
  function metacall(...) returns (bool auctionWon) {
      // ...
-     try this.execute{ /*...*/ }(/*...*/)  returns (bool _auctionWon, uint256 winningSolverIndex) {
-         auctionWon = _auctionWon;
-         (/*...*/) = _settle({winningSolver: auctionWon ? solverOps[winningSolverIndex].from :
↪   msg.sender, /*...*/ });
-         emit MetacallResult( /*...*/, auctionWon ? solverOps[winningSolverIndex].from : address(0),
↪   /*...*/);
-     }
+     try this.execute{ /*...*/ }(/*...*/)  returns (uint256 winningSolver) {
+         auctionWon = winningSolver != address(0);
+         (/*...*/) = _settle({winningSolver: auctionWon ? winningSolver : msg.sender, /*...*/ });
+         emit MetacallResult( /*...*/, winningSolver , /*...*/);
+     }
      // ...
  }
```

Possibly `metacall()` could also return `winningSolver`.

Fastlane: Solved in PR 249.

Spearbit: Verified.

5.5.80 An `else` after a `return` or `revert()` isn't necesssary

Severity: Informational

Context: Atlas.sol#L42-L64, Atlas.sol#L146-L147, Atlas.sol#L221-L222, Atlas.sol#L350-L364, GasAccounting.sol#L209-L214, Escrow.sol#L383-L386, Escrow.sol#L452-L473, AtlasVerification.sol#L623-L625, ExecutionBase.sol#L252-L253

Description: An `else` after a `return` or `revert()` isn't necessary. Removing them makes the code shorter and often easier to read. See for example:

```
function metacall(/*...*/) /*...*/ {
    // ...
    if (/*...*/) {
        if (isSimulation) revert VerificationSimFail(uint256(validCallsResult));
        else revert ValidCalls(validCallsResult);
    }
    // ...
}
```

Recommendation: Consider changing the code to:

```
  function metacall(/*...*/) /*...*/ {
      // ...
      if (/*...*/) {
          if (isSimulation) revert VerificationSimFail(uint256(validCallsResult));
-         else revert ValidCalls(validCallsResult);
+         revert ValidCalls(validCallsResult);
      }
      // ...
  }
```

Fastlane: Solved in PR 190.

Spearbit: Verified.

5.5.81 Errors ValidCalls and VerificationSimFail use different pattern

Severity: Informational

Context: Atlas.sol#L42-L65, AtlasErrors.sol#L32-L37

Description: The error ValidCalls(ValidCallsResult) uses a different parameter than VerificationSimFail(uint256 validCallsResult), while the supplied value is the same. This could use the same pattern.

```
function metacall(/*...*/) /*...*/ {
    (/*...*/ , ValidCallsResult validCallsResult) =
↪   IAtlasVerification(VERIFICATION).validateCalls(/*...*/);
    if (/*...*/) {
        if (/*...*/) revert VerificationSimFail(uint256(validCallsResult));
        else     revert ValidCalls(validCallsResult);
    }
// ...
}
```

```
contract AtlasErrors {
    error VerificationSimFail(uint256 validCallsResult);
    error ValidCalls(ValidCallsResult);
}
```

Recommendation: Consider changing the code to:

```
  function metacall(/*...*/) /*...*/ {
    // ...
-     if (/*...*/) revert VerificationSimFail(uint256(validCallsResult));
+     if (*/...*/) revert VerificationSimFail(validCallsResult);
      // ...
  }
```

```
  contract AtlasErrors {
-      error VerificationSimFail(uint256 validCallsResult);
+      error VerificationSimFail(ValidCallsResult);
  }
```

Fastlane: Solved in PR 252.

Spearbit: Verified.